



UNIVERSITY OF TRENTO

Department of Information Engineering and Computer Science

Master's Degree in
Computer Science

FINAL DISSERTATION

MICRO-ID-GYM: A TOOL TO SUPPORT SANDBOXING AND AUTOMATED PENTESTING OF IDENTITY MANAGEMENT PROTOCOLS

Supervisor

Silvio Ranise

Co-Supervisor

Andrea Bisegna

Roberto Carbone

Student

Giulio Pellizzari

Academic year 2019/2020

Contents

Abstract	3
1 Introduction	4
1.1 Context and Motivation	4
1.2 Problem	4
1.3 Contributions	4
1.4 Structure of the Thesis	5
2 Background	6
2.1 Identity Management Protocols	6
2.2 SAML	7
2.3 OAuth/OIDC	7
3 State of the Art	9
3.1 Methodology	9
3.2 Formal Analysis	10
3.3 Automated Tools	10
3.3.1 General Purpose Tools	11
3.3.2 Tools for OAuth/OIDC	12
3.3.3 Tools for SAML	15
3.3.4 Tools for OAuth/OIDC and SAML	16
3.3.5 Sandboxing and Education	16
3.3.6 Considerations	17
4 Micro-Id-Gym	20
4.1 Architecture	20
4.2 Backend	20
4.2.1 Client and Identity Provider Repositories	21
4.2.2 STIX Repository	22
4.3 Dashboard	23
4.4 Frontend	24
4.4.1 Proxy	24
4.4.2 MSC Drawer	24
4.4.3 STIX Visualizer	25
4.4.4 Pentesting Tools	25
5 Implementation	31
5.1 Backend	31
5.1.1 Client and Identity Provider Repositories	31
5.1.2 STIX Repository	32
5.2 Dashboard	33
5.3 Frontend	37
5.3.1 Proxy	37
5.3.2 MSC Drawer	40

5.3.3	STIX Visualizer	42
5.3.4	Pentesting Tools	43
5.4	Usage	45
5.5	Considerations and Limitations	45
6	Use Cases	48
6.1	Pentesting an IdM Protocol in the Wild	48
6.2	Pentesting an IdM Protocol in a Sandbox	50
6.3	Hands-on Pentesting Experience at the University	52
7	Conclusion	54
	Bibliography	56

Abstract

Thanks to spread of web applications and the digitalization of our society people started using more and more frequently online services. These services usually require the registration of a user account to access the benefits they offer. Password-based authentication is one of the most common techniques used to verify user identity but, considering the large number of online services the user needs to access, this authentication mechanism forces people to remind lots of passwords. For this reason, even if best practices discourage that, people tend to reuse the same password across multiple accounts thus exposing themselves to a higher level of risk. To tackle this issue, Single Sign-On (SSO) has been introduced. SSO is an authentication schema allowing the user to access different services using the same set of credentials. In this thesis, we focused on the protocols implementing this authentication schema where a Client server relies on a trusted third-party server, the Identity Provider, for user authentication. These protocols have been identified as Identity Management (IdM) protocols. A system implementing an IdM protocol is hereafter referred as IdM systems. In this work, we focus on two of the most known IdM protocol: SAML 2.0 SSO and OAuth 2.0/OpenID Connect. Since several vulnerabilities on the implementation of these protocols have been detected, we investigate the state of the art to discover if there were any tools to assist system administrators and developers in the implementation of IdM protocols. As the first contribution, we provide a comprehensive analysis of the actual state of the art of the (semi-)automated pentesting tools to detect vulnerabilities in IdM systems. Using what we learned from the state of the art, we design and implement Micro-Id-Gym (MIG), the main contribution of this thesis. MIG is a tool to support the creation of sandboxes containing IdM system and perform automated pentesting of IdM protocols. MIG consists of three main parts: MIG Backend, MIG Frontend and the dashboard. MIG Backend contains a set of instances of Client and Identity Provider used to recreate IdM system in a sandbox and a repository of Cyber Threat Intelligence information. MIG Frontend provides the tools to support and automate the pentesting activities of IdM protocols. The dashboard is used to set up an IdM system in a sandbox and to configure MIG Frontend tools.

As the last contribution of this thesis, we describe three use cases that prove the effectiveness of MIG. In the first one, we used MIG to test an IdM system in the wild. We collaborate with an important Italian Identity Provider that implements strong authentication required by PSD2 standard to assess the security of its service. We discovered three relevant vulnerabilities and some misconfiguration on the tested IdM system. In the second use case, we tested an industrial IdM system belonging to Italian National Mint and Printing House. We used MIG during the development process to spot vulnerabilities before the IdM system deployment and we discover two misconfiguration and a vulnerability potentially leading to Cross-Site Request Forgery attacks. Finally, in the third use case, we propose to use MIG for a hands-on experience of pentesting IdM protocols during a security class at the university. After the execution of an IdM system in a sandbox, the students perform pentesting activities to learn IdM protocols, the vulnerabilities they might suffer, which attacks can exploit them and the countermeasures that can be implemented.

1 Introduction

1.1 Context and Motivation

In recent years the digitalization of our society has moved faster and a lot of tasks that people were used to do manually, now can be performed with some clicks on the screen of their laptops. For instance, if ten years ago people should have taken the car and reach a supermarket to do their groceries, thanks to the spread of web applications, today they can buy the food they need directly from their sofa. As for the groceries, lots of other services have been digitalized and are now accessible from Internet. However, to identify the people using these services, each online platform requires the registration of a user account. The account contains user information and can often be accessed by a combination of username and password.

The increasing number of online services forces the user to create a lot of accounts [1], to manage several passwords and, as consequence, to reuse some of them [2]. To reduce password fatigue and improve the user experience a new authentication schema called Single Sign-On (SSO) has been introduced. It allows the user to access different services using the same username and password. Doing this, SSO improved the usability of web applications leading to fast adoption of this authentication schema.

Nonetheless, if on the one hand security risks due to the use of the same password for multiple accounts is decreased, on the other SSO introduced other risks. For instance, if an attacker steals user credentials, he can access all the services linked to those credentials. Since this type of authentication schema is adopted both in enterprises and public administration contexts using protocols such as SAML2.0 SSO (SAML) [3], OAuth 2.0 (OAuth)/ OpenID Connect (OIDC) [4, 5], we decided to perform a deep study of the security risk users using these systems might face.

1.2 Problem

Several approaches to the analysis of SSO standard have been adopted. On one side we have formal analysis where mathematical models are used to check if the protocol specification guarantees adequate security to the user. This approach is important because it led to the discovery of significant flaws even before the protocol is implemented [6, 7, 8, 9].

However, further researches proved that often there is a gap between the specification of a SSO protocol and its implementation which results in several vulnerabilities [10, 11, 12, 13, 6, 14]. Developers have to deal with heterogeneous technologies each one with some implementation details that are not taken into consideration by formal models. Furthermore, the process of discovering implementation errors is difficult and very time consuming. Testers have to analyze manually the SSO system looking for flaws. For these reasons, the implementation of SSO is notoriously error-prone even for security expert having strong knowledge in the field.

To address this issue, several (semi-)automatic tools have been developed. These tools aim to fasten the pentesting activities to discover implementation issues and streamline the process of deploying secure SSO systems. However, none of the available tools provide a comprehensive set of test for both SAML and OAuth/OIDC. Additionally, most of the existing tools are designed to perform penetration testing on real scenario on the Internet and this is a problem for organizations such as banks or public administration which can not test directly their production environments but need to check their system security in a isolated environment without the risk of harming their business or systems linked to the tested system.

1.3 Contributions

The contributions provided in this thesis are:

- a **comprehensive analysis** of the state of the art of (semi-)automatic pentesting tools for

IdM protocols. The analysis starts from tools targeting general authentication scheme, moving then to more specific ones targeting OAuth/OIDC and SAML. To complete the analysis we also considered those tools providing features to test systems in an isolated environment and including an educational purpose among the possible uses.

- **Micro-Id-Gym (MIG)** which includes:

- a training environment with **a set of C and IdP instances** of some of the most common technologies that the user can exploit to create a customized implementations of IdM protocols and execute them into a **sandbox**
- **a repository with Cyber Threat Intelligence information** of vulnerabilities, threats and mitigations of IdM protocols.
- **MSC Drawer**, a tool to draw the message sequence chart of the authentication flow between the user browser and the IdM system under test, and to inspect the content of the exchanged messages.
- **STIX Visualizer**, a tool to represent in a graph structure Cyber Threat Intelligence information related to the authentication flow displayed in the MSC Drawer.
- **two pentesting tools** performing automated tests on the system under test. The provided tests include *web application security* tests targeting vulnerabilities common to all web applications, *compliance* tests verifying if the IdM system is compliant with the specifications defined in the IdM protocol standard and *IdM protocol* tests targeting vulnerabilities of the IdM protocol implementation that are not covered by compliance tests.

- **three reference use cases** describing possible applications of MIG in both industrial and educational scenarios. We propose two applications in an industrial scenario, one where we used MIG to test a real deployment of an important Italian company, and the other where we run the IdM system of the Italian National Mint and Printing House in a sandbox and we tested it in an isolated environment. Concerning the educational scenario, we propose an application of MIG in a university context for a hands-on pentesting experience on IdM protocols.

1.4 Structure of the Thesis

The thesis is organized as follow:

Chapter 2 presents a brief overview of SSO, introduces and explains the concept of IdM protocols and illustrates SAML and OAuth/OIDC, the IdM protocol we focused on in the rest of the thesis.

Chapter 3 describes the problem we analysed, the methodology we followed and provides a comprehensive analysis of the pentesting tools available in the state of the art. In the analysis, we also include tools to create isolated testing environments and tools for educational purposes. To conclude the chapter we present a comparison between the existing tools.

Chapter 4 provides a high level overview of MIG, describing its main component, MIG Backend MIG Frontend and the dashboard with their respective structure.

Chapter 5 describes the technologies used to implement MIG Backend, MIG Frontend and the dashboard, explaining the implementation details and justifying our choices. We conclude the chapter discussing advantages and limitations of our implementation.

Chapter 6 presents the three use cases. The first one describes the use of MIG to test a production IdM system of a famous Italian company. In the second we use MIG to run an IdM system of the Italian National Mint and Printing House in a sandbox and we tested it using MIG pentesting tools. Finally, in the third use case, we explain how MIG can be used at the university for a hands-on pentesting experience on IdM protocols.

Chapter 7 contains final considerations about the work that has been done in the thesis and future work that can be carried out starting from what we have proposed.

2 Background

The process towards the digitalization of our society has generated a huge increase in the development of web applications. Nowadays the Internet provides lots of services to simplify and fasten several aspects of everyday life. For instance, people can rent a bike from their laptops, do the groceries from their office, and pay for everything with their preferred online payment service. However, this multitude of services forces users to create a huge number of accounts. In the following sections, an overview of the issues derived from managing a lot of credentials together with a possible solution to support the user is presented. Based on that, the definition of IdM protocols is provided and two of the most known are introduced.

2.1 Identity Management Protocols

To access the services provided by the majority of web applications, people have to register an account on these platforms. In most cases, unique usernames and passwords have to be chosen and then used every time the user wants to access the service. However, even if good practices and guidelines about choosing and managing account passwords are provided [15], the fatigue of dealing with a large number of accounts (23 on average for a common person [1] that scales to 191 for businesses [16]) makes users less prone to follow these guidelines. According to Google, 65% of users reuse the same password across multiple (and in some cases all their) accounts [17]. On average, the same password is used until 14 times and it does not concern only personal accounts. 73% of people adopt the same password for both business and personal accounts thus allowing attackers to exploit compromised private credentials to access also corporate accounts. The most worrying fact is that even if 91% of people claim to understand the risks related to bad password management behaviours, only 32% do something to make their passwords more secure [2].

To reduce the large number of credentials needed, a new authentication mechanism called SSO has been introduced. This authentication schema allows the user to access different services using the same credentials. Doing this, password fatigue is reduced and user experience improved. In a SSO scenario, there are three main entities involved: the user agent, the IdP, and the C. The user agent usually is the browser used by the user to navigate through the Internet. The IdP is the service responsible for the authentication of the user and the C is the service the user wants to access. Therefore, on the contrary of common websites that have to implement by themselves both the service they are offering and the mechanisms to authenticate users, in a SSO system these tasks are split between C and IdP. The former will focus on the development of the best service it can offer while the latter on providing a secure authentication mechanism (e.g. multi-factor authentication). Doing this, C and IdP together provide a higher quality and a higher security service.

A SSO scheme (Figure 2.1) usually works as follows:

1. the user tries to access a C resource that is protected
2. the C asks IdP for user identity verification
3. the IdP performs the user authentication
4. once verified the user identity, the IdP sends user identity data back to the C
5. the C uses the data contained in the response to decide whether granting or denying the permission to access the requested resource to the user

The steps illustrated in Figure 2.1 are a high level overview of the interactions performed. More precise details about each step are provided by the protocols implementing this authentication schema.

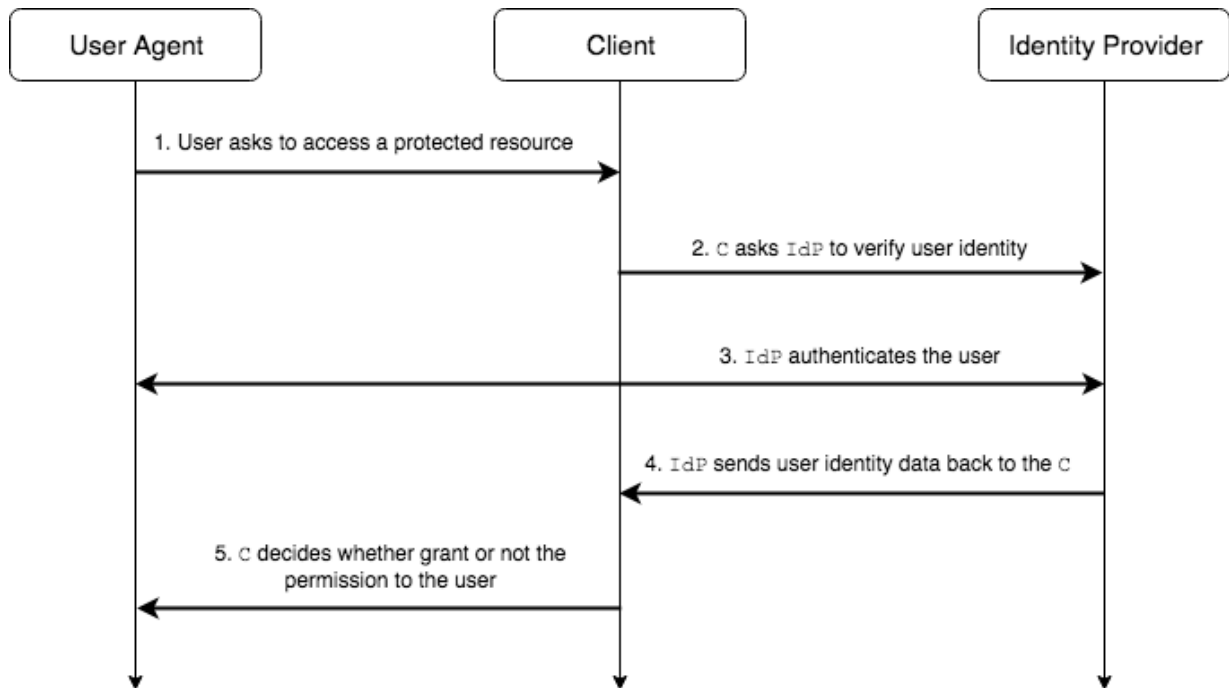


Figure 2.1: Single Sign-On scheme

This type of protocols for identity management where a C server relies on a trusted third-party server, the IdP, for user authentication are hereafter identified as IdM protocols. A system implementing an IdM protocol is identified as IdM system.

The importance of IdM protocols has increased in recent years and today lots of modern websites support this authentication scheme. Two of most widespread are SAML and OAuth/OIDC.

2.2 SAML

SAML is a standard used to exchange authentication and authorization statements mainly adopted in public administration (e.g. eIDAS in EU¹, SPID in Italy²) and enterprise scenarios. It is based on XML messages called *assertions* exchanged between the entities involved. In the case of an IdM system based on SAML, the entities involved take the names of Service Provider (SP) and IdP. While the latter is the same identifier used in the general description of an IdM protocol, the SP is the equivalent of the C previously presented.

Before starting the authentication process the two entities have to establish a relationship via a process called federation. During the federation SP and IdP exchange metadata, information describing identifiers, endpoint URLs, encryption protocols available, certificates of each entity involved. This information is needed before starting any authentication process.

Once the federation is completed, the IdM system is ready to be used. When a user wants to access a resource on the SP, the SP forwards an *AuthRequest* to the IdP. The *AuthRequest* is a message asking the IdP to verify the identity of the user requesting the resource. At the IdP the user performs the authentication and the assertion is generated. The *assertion* is the message containing the information about the identity of the user that is sent back to the SP. The data contained in this message is used by the SP to decide if the user can access the requested resource.

2.3 OAuth/OIDC

OAuth is one of the most adopted standard to allow third parties applications to access resources of another application on behalf of the user. This process, usually called *delegation*, is based on the consent and involves the following entities: the Resource Owner, that usually is the user, the C that

¹<https://ec.europa.eu/digital-single-market/en/trust-services-and-eid>

²<https://www.spid.gov.it/>

access the resources on user behalf, the Authorization Server (AS) which generates the token and the Resource Server (RS) which stores the protected resource the C want to access.

The authorization flow starts when a user tries to access a resource on the C. At that moment, the C sends an authorization request to the AS. Once authenticated at the AS, an authorization form asking to grant the C with the permission to access the needed resourced is prompted. If the user agrees, a *token* is generated and sent back to the C. The C uses this token to access the protected resource on the RS.

However, as it is, OAuth provides only authorization. Authentication can be achieved with OIDC, a layer added on top of OAuth providing profile information about the person that is logged in at the AS. Since this new layer adds information about the user identity, together with the token containing authorization information, the AS generates another token called id token containing identity information. As for the OAuth flow, the two tokens generated by the AS are used by the C to access the requested resource at RS.

As happened for SAML, also in case of OAuth/OIDC the names of the involved entities are different with respect to the general IdM protocol definition previously proposed. Indeed, if C represents the same entity, AS and RS together play the role of the IdP.

3 State of the Art

The improvements in usability introduced by IdM protocols led to the fast adoption of this authentication schema by a large number of web applications around the Internet. The use of only one password to access different services has improved the web applications usability hence meeting the needs of users. Nevertheless, together with the improvements in usability, IdM protocols have brought some risks that security experts have to keep into consideration. It is true that a user can access multiple services with the same username and password but an attacker gaining access to the user credentials can do the same, exposing the victim to severe damages. However, credentials theft is only the peak of the iceberg since a lot of IdM systems suffer implementation vulnerabilities that if exploited could be really dangerous. For this reason, IdM systems have been analyzed by security experts and several flaws have been detected. The following sections present the methodology we followed and the different types of approaches to the analysis of IdM protocols.

3.1 Methodology

The research we conducted followed a qualitative strategy to achieve a deep understanding of *the different approaches applied to the analysis of IdM protocols*. A quantitative approach based on numeric data would not have been suited for our purpose since we do not need statistics and measures about IdM protocols. We need more in-depth information about the most common approaches, their advantages and their limitations, and considerations or suggestions of researchers that previously explored this research topic.

Therefore, to understand more about our research topic, we started the analysis from literature. We used Google Scholar¹ as a search engine to gather an initial list of papers reporting studies on IdM protocols. However, since our definition of IdM protocols does not match with the definition of identity management commonly used in literature, we had to start from a broader view. Our purpose is to find studies reporting analysis of IdM protocols that led to the discovery of unnoticed or uncovered vulnerabilities, therefore we started looking for general vulnerability scanners presented in literature to then refine the results choosing only those that fit our research scope. To discriminate whether an article can be considered in our research scope, we read its abstract and we evaluate if the discussed arguments concern IdM protocols. In particular, we focused on approaches that evaluate the security of IdM protocols and report novel vulnerabilities of IdM systems. To have more relevant information, we ordered the results by pertinence. According to Google Scholar, pertinence takes into account the full text of the article, where it was published, the authors and how often it has been cited by other scholarly literature. We considered these reasonable criteria to order the articles to obtain the most relevant works.

Given that, the first keywords we used for the research were “**vulnerability scanner**” and “**web scanner**”. We got almost 90000 and 30000 pages of results and we inspected the first ten pages of both searches. However, we noticed that the reported articles were too vague for our research purpose therefore we refined the keywords. We looked for “**vulnerability scanner ss**” and out of the almost 4000 pages, we dove into the first twelve but this time we discover nine results in our research scope. We decided to stop at page twelve because the more we move forward in the pages, the more the results were going out of our scope. We kept this termination criterion also for the subsequent researches. Since these keywords provided interesting results, we expanded the acronym sso into single sign-on and we repeated the research. The number of pages this time was higher (10000) and we discovered six other papers that fit our research scope. We also tried with a more generic keyword set formed by the words “**black box web vulnerability scanner**” and out of the

¹<https://scholar.google.com/>

ten pages analyzed, we identify four potentially interesting results. To conclude we tested the keywords “**vulnerability scanner**” together with the name of the IdM protocols presented in Chapter 2. The research for “**saml vulnerability scanner**” provided no interesting results while the ones for “**oauth vulnerability scanner**” and “**openid vulnerability scanner**” highlighted five and seven articles in our scope. With almost thirty papers, we stopped here the research for related works and we started reading them. The cross-references between papers allowed us to identify other articles reporting relevant information about our research scope. At some point, we noticed that all the works cited inside different papers were already in our list therefore we considered it as a reasonable termination point for our papers review. The works done on literature, was then further improved by the addition of other works coming from Github², the world’s leading software development platform, and from an international conference on security³.

The large number of researches on the security of IdM protocols evidenced the relevance of our research topic. During years, researches focused on different aspects of IdM protocols but all works analyzed stress the fact that IdM protocol implementations are notoriously error-prone therefore pursuing the research to find novel solutions to improve the security of IdM system is a challenge acknowledged by the research community. Additionally, the large number of works provide us lots of useful insights. Limitations have been used to learn the critical aspects that we have to keep into consideration when we are analyzing IdM protocols. Instead, we exploit the heterogeneity of design choices proposed as a basis for the construction of our solution. In the following sections, the conducted review is deeply illustrated.

3.2 Formal Analysis

One of the approaches to the analysis of IdM protocols is their formal analysis. Using mathematical models, security experts verify if the specifications describing how the protocol should work are secure or contain some flaws that might lead to an attack. Formal analysis applied to IdM protocols shows that several vulnerabilities were hidden in the protocol specifications.

In 2008, after a formal analysis of SAML, Armando et al. discovered a severe vulnerability in the scheme used by Google services [6]. This flaw allows an attacker controlling a malicious C to impersonate a user at another C. In 2013, using the model checker proposed in [18], Armando et al. discovered another vulnerability on SAML [7]. Since there is no specification forcing that the initial authentication request and the relative assertion have to be transmitted over the same SSL connection, this degree of ambiguity can be exploited by attackers to redirect the victim to a resource different than the one initially requested or to launch Cross-Site Scripting (XSS) or Cross-Site Request Forgery (CSRF) attacks. The same check has been performed for OpenID and it resulted that the vulnerability was also present in that protocol.

A formal analysis of OIDC is presented by Fett et al. in [8]. They formalized OIDC protocol including security measures to prevent previously discovered attacks and attack variants, and demonstrate that the guidelines they propose are efficient and sufficient to ensure central security properties for OIDC. Concerning OAuth, in 2011 Pai et al. [9] used the Alloy framework to discover vulnerabilities in the protocol and they were able to identify a vulnerability in the client credential flow of OAuth. In the same year, Chari et al. [19] presented another framework to study OAuth security called Universal Composability Security Framework. They focused on the authorization code flow and they proved that to guarantee the security of OAuth, all communications must use SSL channel.

3.3 Automated Tools

The works presented in Section 3.2 are all based on formal models that analyze IdM protocol abstractly. Indeed, if theoretically speaking the specification of an IdM protocol can label it as secure, in practice when developers start implementing an IdM protocol, they have to deal with several details such as different technologies and heterogeneous scenarios that are not taken into consideration by formal models. These challenges make IdM systems error-prone even if security experts having strong knowledge in the field are involved in the development process. As a result, numerous vulnerabilities

²<https://github.com/>

³<https://osw2020.com/>

have been detected in these systems [10, 11, 12, 13, 6, 14]. Moreover, the process of discovering these flaws is very time-consuming. Testers have to spend a lot of time on the IdM system under test looking for potential vulnerabilities. These operations are often manually performed which makes the task very long and also prone to tester mistakes. Mistakes that in some cases can be very dangerous. For instance, the lack of compliance with regulations might lead to major security and also legal implication. Therefore, testers need expert knowledge and skills to ensure a proper security assessment of the tested IdM system.

For these reasons, to fasten the pentesting operations and to be sure that tests are performed in the correct way, researchers started the development of (semi-)automatic tools to detect implementation flaws in IdM systems.

3.3.1 General Purpose Tools

The first automated tools developed focused on web applications where the user authenticates on a system to access the provided services. These tools are not tuned to detect vulnerabilities of a particular IdM protocol such as SAML or OAuth/OIDC ones therefore they spot common flaws regarding the implementation of the authentication mechanisms.

The first one belonging to this class of tool is **BLOCK** [20], presented in 2011. The tool developed by Li and Xue works as a proxy and tries to detect state violation attacks. The violations are detected by the comparison of the web traffic generated by the tester machine against a set of invariants defining the expected behaviours. In case an anomaly is detected, BLOCK drops the suspicious message. The expected behaviour is defined by BLOCK in a training phase. During this phase, the tester has to perform several authentication to let BLOCK learn the authentication pattern and define the relative reference invariants. After the learning process, that has to be performed manually, the testing phase on the target system can start. Although tests performed by the authors proved the effectiveness of the tool, it has several limitations. It can only test PHP web applications and look for specific attacks targeting the session. Also, as the authors stated, the completeness and the correctness of the inferred invariants can not be guaranteed making the tool potentially useless.

On the traces of BLOCK, Xing et al. in 2013 proposed **InteGuard** [21]. This tool works very similarly to BLOCK [20] since it is a proxy that inspects web traffic and compare intercepted messages against a set of invariants. Nevertheless, on the contrary of BLOCK that analyzes the communication between two parties (user agent and the PHP website), InteGuard has to deal with a higher level of complexity due to the fact that it tests IdM systems which involve three entities (user agent, C and IdP). In case of InteGuard, to generate the invariants, four different test accounts have to be used and the four different authentication flows must be generated. To define better invariants the generated flow should be as much different as possible. The HTTP traffic observed during these flows is then analyzed, the message parameters are extracted and used to determine precise rules capable to discriminate whether a message is a legitimate one or not. Using these invariants a set of security policies is generated. In case some messages infringe the security policies, InteGuard raises an alarm notifying the user that something suspicious is happening to his system.

The approach of analysing the web traffic generated by the user browser has been adopted by other tools. **BRM Analyzer** [22] focuses on IdM system and extracts protocol specification from the messages generated during the authentication flow. Starting from the fact that an authentication process can be described as a series of browser relayed messages (BRM) exchanged between C and IdP, BRM Analyzer aims to analyze these messages and identify the key elements (i.e. parameters) of the IdM system under test. These elements are then labelled with a type (i.e. int, word, etc.) and a semantic (i.e. user-unique, session-unique, etc.). When the labelling process is finished, BRM Analyzer generates a report providing to the user the understanding obtained through the analysis of the BRM traces. The result of this approach is a logical evaluation of the analyzed flow describing what can be done with each parameter. Unfortunately, even if specifications are extracted automatically, the tester has to manually study the generated report to infer which vulnerabilities might be present on the analyzed IdM solution. For this reason, if on the one hand, the analysis process is a bit speeded up, on the other BRM Analyzer still leaves a lot of work on the tester as confirmed also by the authors that said they spent more time on understanding how each IdM system works rather than on reasoning on the logic model generated.

A year later, Wang et al. implemented **AUTHSCAN** [23]. This tool uses the information obtained from the analysis of HTTP traces and client-side Javascript code to create an initial abstraction of the protocol specification. Then it starts a continuous refining process to obtain the most precise specifications possible. The refinement process exploits both black and white box techniques and stops when no additional information about the IdM system can be inferred. The result of the refinement process is the specification of the IdM protocol of the analyzed IdM system represented using an abstract language called Target Model Language (TML). The generated protocol specifications are then converted from the intermediate language (TML) to applied pi-calculus, a widely used specification language for security protocol. Using automated verification tools, the converted specifications are checked against a set of candidates attacks. If an attack is detected during the specification validation process, it is further verified against an oracle. The real HTTP traffic representing the attack is generated and submitted to the provided oracle which determines whether the attack was successful. This operation is performed to avoid false positives. Using this tool the authors discovered 7 vulnerabilities in real-world applications. However, even if the results are positive, the assumptions the authors made are hard to be met. To make AUTHSCAN works, the tester has to provide to the tool a lot of information: the credentials of two test accounts, the entities relevant to the protocol of the analyzed system (e.g. C and IdP), the API that can be queried to obtain the entities public keys and finally a test oracle that indicates whether authentication process is successful or not. Therefore, although the tests are automated, all these configuration requirements make AUTHSCAN not really usable.

Before diving into the tools specialized in pentesting of a specific IdM protocol, literature proposes another tool to perform vulnerability assessment of general IdM system: **CSRF-checker** [14]. In their study, the authors performed a large-scale analysis of websites with identity management functionalities to verify whether they were vulnerable to authentication CSRF attacks. Authentication CSRF attacks are a particular type of CSRF where, if the attack is successful, the attacker is logged into the victim account or, on the contrary, the victim is logged into an attacker-controlled account. To verify if the target system suffers a vulnerability leading to authentication CSRF, they firstly studied manually the patterns of these attacks and then they distilled some testing strategies to detect the vulnerabilities causing authentication CSRF. The defined strategies have been implemented in CSRF-checker. This semi-automated tool works as extension of the open-source penetration testing tool OWASP ZAP⁴. CSRF-checker exploits the API provided by ZAP to identify the candidate messages, modify them where necessary, and check whether the performed tests have been successful. The results show that CSRF-checker has almost the same efficiency of the manual approach but authors admit that it helps only to reduce as much as possible the manual effort. The generality of the approach not focusing on a particular protocol implies that a minimum set of manual operations have to be done. As the authors stated, if the choice is to develop a tool to test general scenarios, the automation can not be advanced as much as for a tool targeting a specific protocol. For this reason, the tools presented hereafter focus on a specific IdM protocol and try to spot vulnerabilities related to its implementation.

3.3.2 Tools for OAuth/OIDC

Starting from OAuth, the first tool developed to analyze IdM systems using this protocol is **Impersonator** [10]. This tool checks if impersonation attacks performed by sending a stolen or guessed credential to the C's sign-in endpoint are feasible. To do that, it exploits GeckoFX library⁵ to simulate a browser, observe and modify the HTTP messages. Based on the C that has to be tested, Impersonator creates a crafted message and sends it to the C. In case the C does not generate any error, it will be labelled a C as vulnerable to impersonation attacks. Tests performed by the authors proved the tool is useful to discover vulnerabilities leading to this kind of attack since 64 out of 96 tested sites were vulnerable. However, the check for only one vulnerability makes Impersonator very limited to be applied in real-world cases.

To solve this issue, a great improvement in testing OAuth implementations has been brought in 2014 by **SSOScan** [11]. SSOScan is an open-source tool available as web service that performs vulnerability scanner of Cs solutions relying on Facebook IdP to authenticate users. SSOScan simulates attacks and monitors the generated traffic to verify if the authentication has been successfully completed. It

⁴<https://www.zaproxy.org/>

⁵<https://github.com/priyank/GeckoFX>

performs five different attacks: `access_token` misuse, `signed_request` misuse, `app_secret` parameter leakage, and user credentials leakage through referer header or third party scripts accessing the HTML content. The first two tests, `access_token` misuse and `signed_request` misuse, are performed by crafted requests sent to the target C while the last three are deduced by the analysis of the traffic generated during the authentication process. `access_token` misuse and `signed_request` misuse attacks fool the victim C to authenticate the attacker even if he does not have all the needed requirements. However, they can not be performed in the case authorization code flow is used. The other three attacks exploit instead vulnerabilities that provide the attacker access to the victim's confidential information. SSOScan is the first fully automated tool since the user has only to provide the URL of the C he wants to test and SSOScan does everything by itself. It automatically generates two test accounts and uses them to perform the tests. Doing this, user privacy is also protected. The only limitation of SSOScan is that it can only be used for websites relying on Facebook IdP.

OAuth Detector [24] focuses instead on every IdM scenario adopting OAuth. It looks for vulnerabilities leading to CSRF. This tool is a web crawler that examines the source code of a given website to discover OAuth URL links. A link is defined as OAuth URL when the parameters `client_id` and `grant_type` are defined in the URL querystring. When an OAuth URL is found, it is analyzed to determine if it uses the authorization code flow. If it is the case, the `state` parameter is searched. This parameter is the discriminating factor used by the tool to decide if the tested IdM system is vulnerable to CSRF or not. In case `state` is not present, the website is labelled as vulnerable. If no OAuth URL is detected in the main page, every link is followed up to two redirections, and each time the aforementioned detection process is repeated. Using this tool the authors discover that 25% of the domains using OAuth in the Alexa top 10000 sites list⁶ do not properly implement CSRF protection mechanisms. However, they also stated that their estimation is conservative. On the one hand, OAuth Detector does not consider whether an OAuth URL is called via Javascript therefore more domains than the ones identified could be vulnerable. On the other hand, websites might implement non-conventional CSRF protection mechanisms which could make the number of identified websites overestimated.

An approach similar to the ones of [22, 23] is instead implemented by **OAuthTester** [25]. Starting from OAuth RFC [4], OAuthTester constructs a model that abstracts and defines the expected behaviour of the IdM system. Using network traces, the constructed model is further refined to fit behaviour of the IdM system under test. Then, test cases automatically generated by OAuthTester are submitted to the real system. The responses are compared with the ones expected by the generated model to identify anomalies that evidence the presence of vulnerabilities in the IdM system. Deviations from the expected behaviour are reported by the tool as a warning to the user. Additionally, OAuthTester uses these anomalies to refine the constructed testing model. Even if this tool might seem similar to BRM Analyzer [22] and AUTHSCAN [23], on the contrary of them that build their model only on the network traffic analysis, this work adds the protocol specification to the generated traffic to construct a more comprehensive system model. Nevertheless, OAuthTester effectiveness is strictly related to the granularity of the generated model. For instance, authors do not consider the implementation of SSL nor the detailed HTTP header such as cookies therefore the model will not spot vulnerabilities related to the aforementioned aspects causing an implicit side effect on the system security.

In recent years, two other tools for the pentesting of OAuth protocol have been presented: OAuthGuard [13] and OVERSCAN [12]. **OAuthGuard** is a vulnerability scanner and protector for OAuth and OIDC systems using Google as IdP. It focuses on authorization code grant and implicit grant and it is the first solution able to block potential attacks in real-time. OAuthGuard works as Google Chrome extension and protects or warns the user from five different attacks: CSRF attack, impersonation attack, authorization flow misuse, unsafe token transfer and privacy leak. All these attacks are detected by the analysis of the messages exchanged through the browser. For the CSRF attack, OAuthGuard inspects the `redirect_uri` parameter and checks if its value points either to Google or to the C domain. In case the parameter does not match these values, the message is dropped by the extension and the user is notified that a potential CSRF attack has been blocked. Impersonation attack is detected instead when an `access_token` is not bound with any C. In this case, the tool notifies the user with a

⁶<https://www.alexa.com/topsites>

warning telling him that the C might be vulnerable to impersonation attacks. The same happens also in case of authorization flow misuse when more than the only parameter needed (`code`) is submitted by the Cs to Google sign-in endpoints. Concerning unsafe token transfer, OAuthGuard block every message that is transferring `code`, `access_token`, or `id_token` over insecure HTTP channels. Finally, privacy leak protection is applied by blocking pages with third-party contents included in their sign-in endpoints that might steal access information such as `code` or `access_token`. The test on 137 real systems found 69 of them suffering from at least one serious vulnerability in their implementation. This result validates the effectiveness of the tool and makes it one of the best solutions to protect the user during the navigation. The only drawback is that it applies only to IdM system relying on Google as IdP.

This limitation is overcome by **OVERSCAN**. Unlike some of the other existing tools, it is meant to be compatible with any implementation of OAuth. It serves as Burp Suite (Burp)⁷ extension scanning web traffic to discover missing parameters or HTTP headers that might lead to security issues. Using the API provided by Burp, it analyses every message to identify those belonging to OAuth protocol. OVERSCAN supports OAuth implicit and authorization code grant. Headers and parameters of those messages are then checked to verify whether fields allowing the attacker to perform some kind of attacks are missing. `X-XSS-Protection`, `X-Content-Type-Options`, and `X-Frame-Options` are the headers identified by the authors as necessary to prevent attacks like XSS and Clickjacking while `X-CSRFToken` and the `state` parameter are those necessary to protect the user from CSRF attacks. The `redirect_uri` is verified to be over a secure channel (HTTPS). Based on the analysis result, the vulnerable messages are logged in the extension and colored with a color representing the severity of the identified vulnerability. To estimate the level of severity the authors use the Common Vulnerability Scoring System (CVSS)⁸ and group the result in three classes: High, Medium and None. The founded vulnerabilities are also arranged as HTML or PDF report to be more understandable for the user. Therefore, on the contrary of OAuthGuard, OVERSCAN does not provide real-time protection but it summarizes the results in a report. However, both solutions are more than valid as OAuth automated pentesting tools.

The most recent automated tool to test OAuth implementation is **OAuch** [26], a tool presented during the 2020 OAuth Security Workshop [27]. OAuch is a web application acting as malicious client to find security weaknesses. Each test it performs correspond to one security related requirement of the OAuth specification at any requirement level. This means that each *MUSTs*, *SHOULDs* and *MAYs* present in the OAuth specifications represents a test case. Once inserted basic information (e.g. endpoints, client id, etc.) into OAuch interface, the tests are automatically executed. OAuch is able to detect which grants have been implemented (code, implicit, etc.) and testing them accordingly. As for the grants implemented, OAuch tries to guess as many setting as it can. Once the tests are finished, a report page show all the information about the test results. For each test, the user can inspect a log with detailed information.

Concerning OpenID, the first tool for automated pentesting presented in literature is **OpenID Attacker** [28]. Given the fact that OpenID is an open system, every C can choose its own IdP, therefore the authors decided to study a set of attacks that exploits a malicious IdP. They successfully tested four novel attacks against existing OpenID system manually and then implemented OpenID Attacker to automate the pentesting process. OpenID Attacker mainly works as a malicious IdP and runs in three modes: analysis, manual attack, fully automated attack. In the first mode, analysis, OpenID Attacker acts as honest IdP creating valid tokens. This mode is used to gather information about the C such as OpenID version, parameters used, etc. This information is used by the tool during the other working modes. In manual attacker mode, the user manages manually the malicious IdP. He manipulates the messages and observes the outcome. This phase is particularly useful if the tester wants to inspect the behaviour of the tested C with respect to targeted stimuli. Additionally, the attacks performed during the manual mode can be recorded and automatically used in the future to test other OpenID systems. Finally, in fully automated mode, OpenID Attacker sequentially executes the four tests provided by the tool and the tests recorded during the manual mode. The provided

⁷<https://portswigger.net/burp>

⁸<https://www.first.org/cvss/v3.0/specification-document>

attacks, namely token recipient confusion, ID spoofing, discovery spoofing, and key confusion, exploit the malicious IdP to impersonate the victim at an honest IdP. The general idea of these attacks is that the malicious IdP is used by the attacker to steal a valid token generated by the victim. The authors tested the tool against existing OpenID systems and they were able to discover vulnerable C on famous services like Sourceforge and Drupal.

However, OpenID attacker has to be download and manually configured. To overcome these limitations, Mainka et al. proposed **PrOfESSOS** [29]. PrOfESSOS is an open-source implementation for fully automated testing of IdM systems using OIDC and it is provided as web service therefore it does not need any configuration. Following the approach of OpenID Attacker, PrOfESSOS acts as malicious IdP and tests two classes of attack: single-phase and cross-phase. Single-phase attacks are those attacks exploiting an insufficient or missing verification step on a security-relevant parameter that should be performed at a single point. Examples of this class of attacks are ID spoofing, replay attacks, signature bypass. Cross-phase attacks are instead those that can be conducted only if the attacker is able to manipulate messages during multiple phases of the authentication flow. For instance, the IdP confusion attack tested by PrOfESSOS requires a preparation phase where the victim C is fooled by the attacker IdP which provides the C with the same `client_id` of a honest IdP. Once this operation is completed, when the victim tries to authenticate on the honest IdP, PrOfESSOS slightly modifies the parameters exchanged so that the `code` generated by the honest IdP is redeemed at the malicious one. This operation gives the attacker access to `code`, `client_id`, and `client_secret` which can be used to impersonate the victim. The other cross-phase attack provided by PrOfESSOS is malicious endpoint attack and works similarly to the one just described. This tool is probably the best tool to test OIDC scenario. However, as the authors stated, it is not intended to be for large scale analysis such as Alexa top sites. Its main goal is to support developers during the implementation of OIDC systems.

The last tool for automated pentesting of OAuth/OIDC implementation is **FAPI conformance suite** [30]. This suite is an automated tool to perform compliance testing for Financial-grade API (FAPI) [31]. FAPI is a profile built on top of OAuth and OIDC designed for high-risk scenarios such as financial ones. It combines together several security mechanisms provided by OAuth and OIDC to achieve a high-level of security. Its definition has been supported by the OpenID Foundation⁹ and famous companies such as Microsoft and Nomura Research Institute, the largest Japanese consulting firm. To promote interoperability and robust implementations, the OpenID Foundation enabled a certification of FAPI deployments. The certification can be obtained only if the system implementing FAPI successfully passes all the compliance tests of the FAPI conformance suite. Since FAPI profile is built on top of OAuth and OIDC, the conformance suite verifies also the compliance with OAuth and OIDC standards. Therefore, even if this suite is thought to be suited for testing system implementing FAPI, it can be also used to check the compliance with both OAuth and OIDC standards.

3.3.3 Tools for SAML

Some automated pentesting tools have been developed to test also IdM systems implementing SAML. The first ones to approach automated pentesting of this protocol have been Mainka et al. in 2012 when they presented **WS-Attacker** [32]. In the paper, the authors describe a framework to test several XML-specific vulnerabilities (SOAPAction spoofing, WS-Addressing spoofing, XML Signature Wrapping, XML-based DoS attacks, New Adaptive and Intelligent Denial-of-Service Attacks and XML Encryption attacks) but the automated tool they implemented is able to check only SOAPAction spoofing and WS-Addressing spoofing. These two attacks are both based on vulnerabilities of SOAP protocol. To perform the attacks, WS-Attacker expects as input a WSDL¹⁰ file containing the description of the service the tool has to test. Based on the information in the WSDL, WS-attacker generates two requests: one correct to learn the expected behaviour and one tampered to test the vulnerability. The results are then compared and presented to the user. However, the main limitation of WS-Attacker is that, from our experience, SOAP is almost never used nowadays by IdM system implementing SAML since most of them exploit HTTP to transfer authentication messages. Therefore WS-Attacker can be considered useless to test real IdM systems using SAML.

This is not the case of **SAML Raider** [33] and **EsPreSSO** [34]. Both tools work as Burp extension

⁹<https://openid.net/foundation>

¹⁰<https://www.w3.org/TR/wsd1.html>

and perform tests of XML-specific vulnerabilities. SAML Raider verifies if the target SAML IdM system is vulnerable to XML Signature Wrapping (XSW) attacks. In this type of attacks, the attacker modifies XML message structure injecting malicious information which does not invalidate the XML Signature. The attack purpose is to change the message in such a way that the signature is not invalidated and the malicious information inserted is processed by the victim service. If this attack is possible, the attacker might be able to impersonate another user at the C [35]. Additionally, using SAML Raider the user can remove XML signature, edit SAML messages and resign them with custom certificates. EsPreSSO provides instead other features. In the proxy log where all the intercepted messages are collected, EsPreSSO highlights the ones belonging to IdM protocols, decodes in real-time message parameters that are encoded (i.e. **SAMLRequest**/**SAMLResponse**) and implements tests to check the presence of vulnerabilities leading to XML External Entities (XXE) attacks. This class of attacks exploits the possibility to load external files into a XML document to perform different types of attacks. For instance, a famous attack is the Billion Laughs attack [36], a Denial of Service (DoS) targeting the entity that will process the tampered XML message. This class of attack is tested also by another tool called **SAMLyze** [37] but it is actually not available. Nonetheless, even if these tools provide pre-compiled attack vectors, part of the testing process has to be performed manually. When the proxy intercepts a message that might be vulnerable to the attack, the user has to manually select the attack vector he wants to inject and then click a button to proceed with the attack. Based on the outcome of the authentication flow, the user has to discriminate whether the attack has been successful or not. This is the main limitation of the presented tools.

3.3.4 Tools for OAuth/OIDC and SAML

If we look for comprehensive tools for automated penetration testing targeting specific vulnerabilities of both OAuth/OIDC and SAML, the only result we find is **WPSE** [38]. WPSE is a security monitor for web protocols implemented as Google Chrome extension designed to ensure compliance with the intended protocol flow, as well as confidentiality and integrity properties of messages. It requires as input the specification of the protocol flow and a security policy. These two elements are used to evaluate the traffic generated by the user browser and block the navigation in case of anomalies. The tool is able to detect attacks causing protocol flow deviation, secrecy and integrity violation. Protocol flow deviations are those attacks where the user browser skips some protocol messages supposed to be processed or processes messages in the wrong order. For instance, in some CSRF, the victim completed an authentication flow that was not initiated in his browser (e.g. Login CSRF described in [14]). Secrecy violation refers to those attacks where confidential information is unintentionally leaked (i.e. via third-party code) while integrity violation is caused by those attacks not satisfying integrity constraint of the protocol (i.e. `redirect_uri` modified during the authentication flow). However, as the authors admitted, WPSE has still to be improved. For instance, if we consider protocol flow deviation causing CSRF in OAuth, the tool blocks every solution that is not using the `state` parameter. Even if this parameter is indicated as CSRF countermeasure by OAuth specification, it does not necessarily mean the websites do not implement any countermeasure, but WPSE would block anyway the responses from these websites. Improving the way of detecting CSRF as proposed in [13] or raise a warning and let the user decide whether to proceed or not could be feasible solutions.

3.3.5 Sandboxing and Education

All the tools presented until now have a common characteristic: the tests performed to validate the proposed solutions have been executed on real-world IdM systems. This is usually done to demonstrate that the proposed solution is effective not only from a theoretical point of view but also in practice. Moreover, big companies such as Google, Facebook, and Microsoft promote bug bounty programs [39, 40, 41] with prizes for people discovering and reporting security flaws in their systems. However, in some cases, pentesting activities on real-world applications might arise some issues. The first aspect to keep in consideration is the legal aspect: if some companies such as the aforementioned ones appreciate if people test the security of their system, others, such as banks or public administrations, might react in an opposite manner if they discover someone that is trying to penetrate in their systems. This aspect is supported by the fact that some type of attacks such as DoS might have severe repercussion

on the impacted business. Therefore, together with automated pentesting tools, it is important to have an adequate environment to safely assess the security of an IdM system. A possible solution to this problem is sandboxing on a local machine the tested environment so that it runs isolated and, in case of failures, the surrounded system is not harmed. To this end, Dashevskyi et al. proposed **TESTREX** [42], a testbed for repeatable exploits with the following features: packing and running applications with their environments, injecting exploits and monitoring their success, and generating security reports. Instead of using heavy virtual machines running the web application, TESTREX exploits Docker¹¹, a software providing virtualization at the operating system level using containers. Containers are sandboxed images that rely on Linux kernel but are isolated from the operating system. Therefore, the application that has to be tested is loaded into these containers. The exploits that simulate the attacks have to be written by the user starting from a reference class providing basic features. Once the attacks are created, they are automatically executed using Selenium Web Driver¹², a tool that automates browser interactions. The result of each test is added in a report file the user can examine at the end of the testing process to eventually debug the web application and the exploits. As the author suggested, this tool can be used in industrial scenarios but also for educational purposes. Indeed, in addition to the possibility of deploying the user own web application, TESTREX also provides a group of vulnerable web applications to illustrate its usage over a variety of web programming languages. The available testing solutions are partly developed by the authors and partly taken from BugBox [43] and WebGoat [44], two existing environments used for testing and learning common vulnerabilities of web applications and their exploits. **BugBox** is a framework to streamline collection and sharing of vulnerabilities, facilitate security experiments, and educate on the most common vulnerabilities. It provides a set of environments where each vulnerability is coupled to an automated exploit. The full source code of the vulnerable applications is also provided. Using BugBox, the user can reproduce well-known exploits. As for TESTREX, these exploits are automatically executed using Selenium. **WebGoat** is instead an insecure web application used to test and learn common web application vulnerabilities. When the user visits WebGoat, he can choose which vulnerability he wants to learn and the tool guides him during the whole learning process. The vulnerability is described and then, following the provided instructions, the user has to perform the attacks exploiting the described vulnerability.

Talking about educational tools, it is worth the mention of two online services providing learning experience for IdM protocol: **Okta OAuth Playground** [45] and **Google OAuth Playground** [46]. Okta OAuth Playground offers practical examples to learn OAuth grant types while Google OAuth Playground provides a sandbox where the user can play with Google APIs supporting OAuth.

3.3.6 Considerations

Using what we learned from the state of the art just described, we design and develop **Micro-Id-Gym (MIG)** a tool to support the creation of sandboxes containing IdM system and perform automated pentesting of IdM protocols. MIG provides a set of instances of C and IdP used to recreate IdM system in a sandbox and the tools to support and automated the pentesting activities of IdM protocols. In Table 3.1 we report a comparison between all state-of-the-art tools for pentesting and also those providing sandboxing and learning functionalities that we have presented.

As previously done, we divided the tools in three sub-group: *general purpose*, to refer the tools used to test web applications where a user authenticates on a system, *OAuth/OIDC*, to refer the tools targeting system implementing OAuth/OIDC, and *SAML*, to refer the tools targeting system implementing SAML.

The features selected for the comparison are the following:

- **(Semi-)Automated:** whether the tool executes tests automatically or semi-automatically with a small interaction of the user
- **Web Application Security:** whether the tool tests vulnerabilities common to all web applications

¹¹<https://www.docker.com>

¹²<https://www.selenium.dev>

- **Compliance:** whether the tool verifies the compliance with the specifications defined in the IdM protocol standard
- **IdM protocol:** whether the tool tests vulnerabilities of the IdM protocol implementation that are not covered by compliance tests
- **Active tests:** whether the tool modifies the exchanged messages to verify how the system under test reacts to the change
- **Passive tests:** whether the tool analyzes the traffic generated during the authentication flow to discover anomalies
- **MSC:** whether the tool provides the list and/or the message sequence chart of messages exchanged during the authentication flow

Table 3.1: Comparison between MIG and the other state-of-the-art tools

	Pentesting							Sandboxing & Education		
	(Semi) Automated	Web Application Security	Compliance	IdM protocol	Active tests	Passive tests	MSC	Vulnerable instances	Automatic deploy	Learning information
General purpose										
AUTHSCAN [23]	✓	✓	-	-	✓	-	-	-	-	-
BLOCK [20]	✓	✓	-	-	-	✓	-	-	-	-
BRM Analyzer [22]	-	-	-	-	-	-	✓	-	-	-
BugBox [43]	✓	✓	-	-	-	-	-	✓	-	-
CSRF-checker [14]	✓	✓	-	-	✓	-	-	-	-	-
InteGuard [21]	✓	✓	-	-	-	✓	-	-	-	-
TESTREX [42]	✓	✓	-	-	-	✓	-	✓	✓	-
WebGoat [44]	-	-	-	-	-	-	-	✓	-	✓
OAuth/OIDC										
Google Playground [46]	-	-	-	-	-	-	-	-	-	✓
FAPI conformance suite [30]	✓	-	✓	-	-	✓	-	-	-	-
Impersonator [10]	✓	-	-	✓	✓	-	-	-	-	-
OAuch [26]	✓	-	✓	-	-	✓	-	-	-	-
OAuth Detector [24]	✓	-	✓	-	-	✓	-	-	-	-
OAuthGuard [13]	✓	✓	✓	✓	-	✓	-	-	-	-
OAuthTester [25]	✓	-	✓	✓	-	✓	-	-	-	-
Okta Playground [45]	-	-	-	-	-	-	-	-	-	✓
OpenID attacker [28]	✓	-	-	✓	✓	-	-	-	-	-
OVERSCAN [12]	✓	✓	✓	-	-	✓	✓	-	-	-
ProFESSOS [29]	✓	-	-	✓	✓	-	-	-	-	-
SSOScan [11]	✓	-	✓	✓	✓	✓	-	-	-	-
WPSE [38]	✓	-	✓	✓	-	✓	-	-	-	-
SAML										
EsPreSSO [34]	✓	-	-	✓	✓	-	✓	-	-	-
SAMLyze [37]	✓	-	-	✓	✓	-	-	-	-	-
SAML Raider [33]	✓	-	-	✓	✓	-	-	-	-	-
WPSE [38]	✓	-	-	✓	-	✓	-	-	-	-
WS-Attacker [32]	✓	-	-	✓	-	-	-	-	-	-
Micro Id Gym	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

✓ = provided feature; - = feature not provided

- **Vulnerable instances:** whether the tool provides vulnerable instances for pentesting and learning purposes
- **Automatic deploy:** in case the tool provides vulnerable instances, whether the vulnerable instances can be deployed automatically with a simple command or an automatic script
- **Learning information:** whether the tool provides information to learn concepts about IdM protocols and to raise security awareness

From the results the table shows, we can assert that it does not exist a tool capable to provide all the features MIG supports. In the following chapters, all the features provided by MIG are described in detail.

4 Micro-Id-Gym

We identify as IdM protocols the web protocols for identity management where a C server relies on a trusted third-party server called IdP for user authentication. The authentication pattern we focused on starts when a user tries to access a C resource that is protected. In this type of scenario, the IdM protocol rules how the authentication data is exchanged in terms of communication patterns that must be followed and security policy that must be enforced.

SAML and OAuth/OIDC are two of the most known standard implementing this authentication schema. The former, SAML, exchanges authentication messages in XML format and is mainly used in enterprise scenarios while the latter OAuth/OIDC is mostly adopted by common web application. To support the creation of sandboxes containing IdM system and perform automated pentesting of IdM protocols, we propose Micro-Id-Gym (MIG). The support provided by this tool aims to assist system administrators and testers in the deployment of resilient instances of IdM systems, as well as for teaching purposes [47].

In the following sections, the architecture of MIG is presented.

4.1 Architecture

MIG provides a set of tools to automate the pentesting activities of IdM systems. Nonetheless, pentesting activities on production systems might be limited due to the possible impacts on the service, for instance in the case of a DoS, but also for the existing regulations that add constraints to the pentesters. For these reasons, together with the set of automated pentesting tools MIG provides also the possibility to recreate an IdM system in a sandbox. The creation of an isolated environment such as the sandbox where the IdM systems runs prevent attacks and failures from spreading across the system surrounding the sandbox. Therefore, by using MIG the user has the tools to automated pentesting activities both in the wild, which means a real scenario on the Internet, and in a sandbox in the laboratory.

The architecture of MIG is described in the high-level architecture presented in Figure 4.1. MIG has been divided into three main components: MIG Backend, MIG Frontend and the dashboard. MIG Frontend consists of the tools to support and automate the pentesting activities while MIG Backend stores Cyber Threat Intelligence (CTI) information¹ encoded in STIX² format and the instances of C and IdP that can be used to recreate locally a sandbox containing an IdM system. Finally, the dashboard is responsible for the generation of the sandboxes and the configuration of MIG Frontend tools. In addition to these elements, Figure 4.1 shows a component called System Under Test (SUT). It represents the IdM system that will be the target of the pentesting activities, either it is a real one in the wild or a sandboxed one in the laboratory.

In the following sections, more details about MIG Backend, the dashboard and MIG Frontend are provided.

4.2 Backend

MIG Backend component stores a set of instances of C and IdP used to recreate IdM systems in a sandbox and CTI information useful to assess the severity of a vulnerability. As presented in Figure 4.1, it is formed by three repositories, Client, Identity Provider, and STIX, containing respectively the available C instances, the available IdP instances and a collection of CTI information encoded in STIX format.

¹<https://www.enisa.europa.eu/events/cti-eu-event/cti-eu-event-presentations/cti-information-sharing/>

²<https://oasis-open.github.io/cti-documentation/stix/intro>

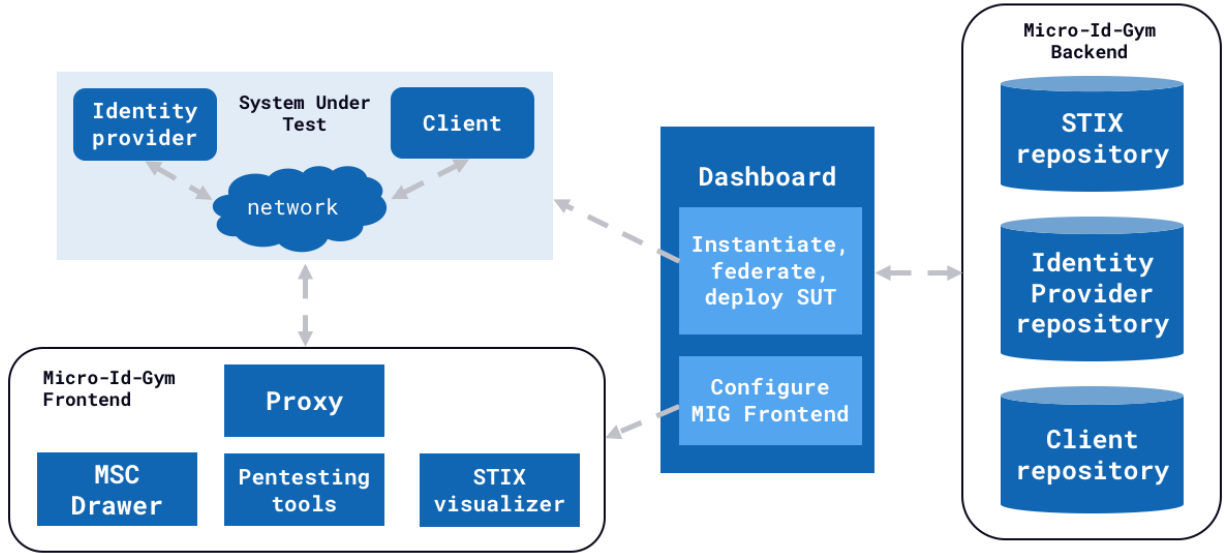


Figure 4.1: MIG High Level Architecture

The reasons for providing a set of existing C and IdP instances are multiple. The available technologies have been selected among the most famous implementation recognized nowadays [48, 49, 50] to provide realistic testbeds to system administrators and testers. Using the dashboard (Section 4.3), they can pick the technologies used in their production system from MIG available ones, configure and deploy a local replica of their IdM systems, and then test it. Doing this, system administrators and testers obtain significant results, since they have tested a copy of their production systems, and do not risk any type of damage to their business. They can also exploit the sandboxes IdM systems to increase their knowledge about their production system behaviors. In addition to that, system administrators and testers have the possibility to explore how other implementations interact with the one they are using and discover whether adopting a different solution could improve the IdM system they are currently using.

However, MIG does not only support business operations. Students can use the different implementations available in the tool to put hands on concepts such as authentication and authorization, and discover vulnerabilities and threats that might affect IdM systems.

4.2.1 Client and Identity Provider Repositories

The list of the Cs currently available in MIG is reported in Table 4.1. All Cs are developed in Java. Concerning OAuth/OIDC, we provide two simple webapps with the basic feature necessary to interact with Keycloak³ and MITREid⁴ IdP.

Concerning SAML, we relied on SAML extension of Spring framework⁵ for the development of the available versions. We provide six instances as testbeds for several case studies reported in literature. The *Base* version provides basic functionalities to interact with the IdP. *c14n algorithm* version implements SAML client that applies the *c14n* canonicalization method before signing the XML document. This version has been added because Duo team found that in some cases this canonicalization format can be used to carry out impersonification attacks [51]. *DTD enabled* version implements SAML client that allows the inclusion of external entities in the XML message. Several attacks exploiting this configuration are reported in literature [29]. One of the most famous is billion laughs attack [36] that leads to a DoS of the target machine. *DTD disabled and c14nWithComments algorithm* is the version where the two aforementioned vulnerabilities are fixed. External entities are not allowed and the canonicalization method adopted is *c14n#WithComments*. The former countermeasure prevents DoS attacks such as billion laughs while the latter includes comments in the canonicalization thus preventing the impersonification attack reported in [51]. The last two versions, *RelayState validation*

³<https://github.com/keycloak>

⁴<https://github.com/mitreid-connect/>

⁵<https://spring.io/projects/spring-security-saml>

enabled/disabled, differ from the action performed on the **Relaystate** parameter. As reported in [6], the **Relaystate** parameter can be used as launchpad for XSS and CSRF attacks. If the C does not validate the **Relaystate** parameter value against dangerous characters (i.e. <, >, ", etc.) the attacks reported by Armando et al. [6] might be carried out. In case it is validated, these attacks are prevented.

The list of the IdPs currently available in MIG is reported in Table 4.2. All IdPs are developed in Java. Concerning OAuth/OIDC, we provide two versions for both Keycloak and MITREid IdP. One version validates **redirect_uri** parameter while the other does not perform any verification. We recreated these two versions because missing **redirect_uri** validation vulnerability is a common vulnerability than has been also discovered on famous platforms like Facebook, Microsoft, and Github [55]. As reported in [56], an attacker can exploit this vulnerability to redirect the user to a fraudulent location controlled by him thus exposing the victim to attacks such as XSS, CSRF and **code** theft. Concerning SAML, we adopted Shibboleth⁶ as reference technology. We provide six different versions, one of these also with the validation of the **Relaystate** parameter enabled. From the analysis of the six versions proposed, we noticed that none of them performs the validation of the **Relaystate** parameter. Even if the standard does not force the IdP to perform the validation, with the modified version we developed, we want to stress the fact that validation performed by the IdP should be a common practice. If validation is implemented, this parameter can not be used anymore as launchpad for XSS and CSRF attacks [6] and all the Cs federated with the IdP is consequently secure. Otherwise, if this is not the case as for common Shibboleth versions, the responsibility for the validation is left to the C.

4.2.2 STIX Repository

STIX repository contains a set of CTI information encoded in STIX format. The acronym STIX stands for Structured Threat Information Expression and it is a format defined by OASIS CTI TC⁷ to

⁶<https://wiki.shibboleth.net/confluence/display/IDP30/Home>

⁷<https://www.oasis-open.org/committees/cti>

Table 4.1: Collection of C instances.

Technology	Version	Protocol
Spring [52]	Base	S
Spring [52]	c14n algorithm	S
Spring [52]	DTD enabled	S
Spring [52]	DTD disabled and c14nWithComments algorithm	S
Spring [52]	RelayState validation enabled	S
Spring [52]	RelayState validation disabled	S
KeyCloak [53]	Simple Webapp	O
MitreID [54]	Simple Webapp	O

O: OAuth/OIDC; S: SAML.

Table 4.2: Collection of IdP instances.

Technology	Version	Protocol
Shibboleth IdP [57]	3.3.3	S
Shibboleth IdP [57]	3.3.3 with RelayState sanitization enabled	S
Shibboleth IdP [57]	3.3.2	S
Shibboleth IdP [57]	3.3.1	S
Shibboleth IdP [57]	3.3.0	S
Shibboleth IdP [57]	3.2.1	S
Shibboleth IdP [57]	3.2.0	S
MitreID [54]	1.3.3 without redirect_uri validation	O
MitreID [54]	1.3.3 with redirect_uri validation	O
Keycloak [53]	10.0.1 without redirect_uri validation	O
Keycloak [53]	10.0.1 with redirect_uri validation	O

O: OAuth/OIDC; S: SAML.

exchange CTI information. As reported by ENISA, CTI information is any information that can help an organization to identify, assess, monitor, and respond to cyber-threats⁸. The CTI information we provide consists of vulnerabilities, attacks and countermeasures related to SAML. During a previous work of one of our colleague [58], this information has been gathered from *OASIS Security and Privacy Considerations for the SAML V2.0* [59], the security threats affecting IdM system based on SAML that are reported in literature and online sources such as blogs. Following OWASP Web application penetration testing guide⁹, this information has been carefully analyzed. The results identify four main vulnerability classes:

- Incorrect generation: vulnerabilities whose cause is the improper creation of Requests or Responses during the SAML process for authentication
- Incorrect checks: flaws related to the verification of the exchanged SAML messages
- Missing CSRF protection: vulnerabilities that do not provide proper protection measures to avoid CSRF attacks
- Missing XML validation: issues related to the specific verification of XML document

The analysis identifies 14 vulnerabilities and 36 attacks targeting Cs and 11 vulnerabilities and 29 attacks targeting IdP. Each vulnerability is described together with the attacks which might exploit it and the countermeasure that should be applied to patch the flaw. Additionally, for each attack the security property violated (CIA¹⁰) and a qualitative risk level are provided.

This data has been then encoded in STIX format to be easily managed by our tools. Particularly, each vulnerability, attack and mitigation has been mapped semantically as STIX *objects* and linked together using *relationships*.

We think this information provides fundamental support both for enterprises and students using MIG. On the one hand, companies will use CTI data to prevent or mitigate threats, assess the risk of discovered vulnerabilities and support the risk management process. On the other, students will learn which are the vulnerabilities affecting IdM systems, how attacks exploit these flaws to cause damages and which mitigations can be applied to fix the discovered problems.

4.3 Dashboard

The dashboard is the component used to set up an IdM system in a sandbox and to configure MIG Frontend. The configuration of MIG Frontend concerns the proxy, the MSC Drawer and STIX Visualizer (Section 4.4.1, 4.4.2 and 4.4.3 respectively). Through the interface the user can customize the ports of these three tools will be executed.

Once selected the protocol the user wants to test, if he decides to create a local IdM systems in a sandbox, he has to pick C and IdP he wants to use to create the IdM system that will run in the sandbox. At the moment of C and IdP choice, the dashboard will show which attacks the selected entities might suffer. The user can also insert customized credentials to authenticate on the IdP and configure the ports where C and IdP will run.

When this information is provided, the process of creation of the sandboxes that will contain the IdM system and the configuration of MIG FE tools is straightforward. The files to configure the provided pentesting tools are customized based on the user preferences. In case the user chose to recreate an IdM system in a sandbox, the chosen instances are retrieved and configured following the directives inserted in the dashboard. The relationship between C and IdP is established and the files necessary for the IdM system deployment are generated.

In addition to these features that concern configuration and deployment of instances and tools provided by MIG, our tool offers also the possibility to extend the environments available in MIG. The user can upload either a custom IdM system he wants to test or a custom tool he wants to add to MIG

⁸<https://www.enisa.europa.eu/events/cti-eu-event/cti-eu-event-presentations/cti-information-sharing/>

⁹<https://owasp.org/www-project-web-security-testing-guide/>

¹⁰<https://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA>

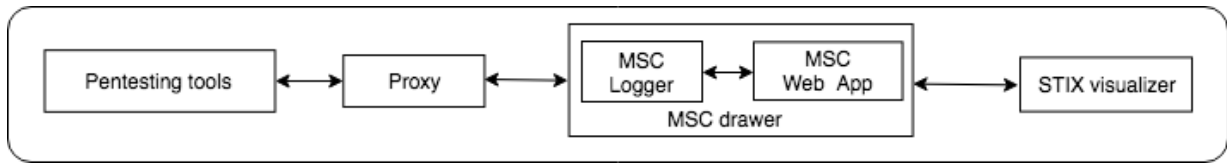


Figure 4.2: MIG Frontend architecture

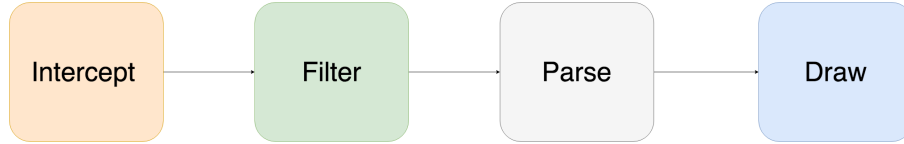


Figure 4.3: MSC Drawer working blocks

pentesting tools. In both cases, the uploaded artefact has to be compliant with the guidelines of our environment. These guidelines are detailed in Chapter 5.

4.4 Frontend

MIG Frontend is used to support and automate the pentesting activities on the SUT. As presented in Figure 4.2, it consists of four elements: the proxy, MSC Drawer, STIX Visualizer, and the pentesting tools. These elements work together to assist user pentesting activities.

4.4.1 Proxy

The proxy element is used to intercept the HTTP traffic between the user agent and the SUT. The intercepted messages can be inspected, manipulated, replicated and if necessary dropped using a set of APIs the proxy provides. These APIs are the ones exploited by the pentesting tools to execute the automated tests. The proxy communicates with the MSC Drawer thanks to an extension called MSC Logger. Figure 4.3 shows the tasks performed to draw the messages in the MSC Drawer. MSC Logger is responsible for the first three blocks, namely *Intercept*, *Filter* and *Parse*, while the last one, *Draw*, is performed by MSC Drawer.

Intercept block is an intrinsic feature of the proxy that works as intermediary between a client and a server. The messages intercepted by the *Intercept* block are the inputs of the *Filter* block. Even if the task of this block might seem easy, in reality, it is quite complicated. IdM protocol specifications describe which are the mandatory messages that have to be exchanged between the entities involved in the authentication process but there is no specification about what happens around those messages. Each implementation might add personalized requests (e.g. images, advertising, etc.) sent between two messages that, according to the IdM protocol specification, are supposed to be subsequent. Therefore, isolating the messages belonging to the IdM protocol standard flow is a complex task. To solve it, we decided to follow a white-list approach. We define some filters that describe the messages belonging to the IdM protocol and only those messages matching these filters are forwarded to the following block. The messages not belonging to the IdM protocol are discarded by the *Filter* block. The *Parse* block takes the raw IdM protocol messages arriving from the *Filter* block and extracts the information needed using the APIs provided by the proxy. In case the user has configured the proxy to decode parameters such as `SAMLRequest` that are transmitted encoded, this block performs the required decryption. The data is then formatted in a proper way and sent to the MSC Drawer through its API. MSC Drawer will use them to draw the message sequence chart.

4.4.2 MSC Drawer

As mentioned in the previous section, the MSC Drawer is responsible for the last of the tasks presented in Figure 4.3, the *Draw* block. The MSC Drawer uses the data pushed by the MSC Logger to draw the message sequence chart of authentication flow performed on the SUT (Figure 4.4). The message sequence chart provides a graphical representation of the messages exchanged and it represents the first step to assist the user pentesting activities. Indeed, the comparison between the message sequence chart drawn in the MSC Drawer and the one defined by the standard highlights at first glance whether

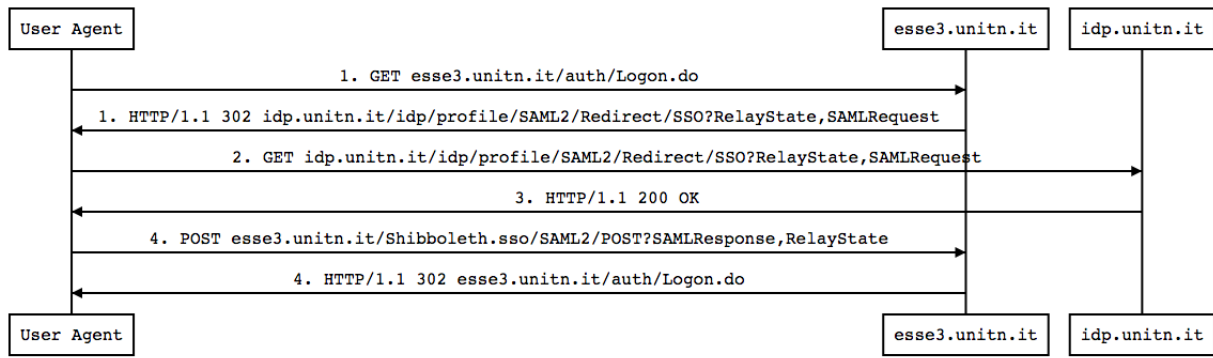


Figure 4.4: Example message sequence chart of an IdM system implementing SAML

the SUT is compliant with the standard or not. Therefore the user can spot something anomalous just by looking at the graph.

However, this is not the only feature provided by MSC Drawer. The user can dive into headers, parameters, and body of the displayed messages to inspect them in detail. To increase the usability of MSC Drawer, it is also possible to search in the displayed messages. The search can be general, on the whole content of the messages, or specific on a particular field such as headers or parameters. This feature is useful when the user wants to quickly verify the value of a header (e.g. `X-FRAME-OPTIONS`) or the presence of a parameter that is used to prevent an attack (e.g. `state` parameter to prevent CSRF in OAuth/OIDC).

Finally, the MSC Drawer provides an interface to lookup CTI information related to the message sequence chart displayed.

4.4.3 STIX Visualizer

STIX Visualizer provides a graphical representation of CTI information encoded as STIX objects and the relationships between them. Each object is represented as a node while the relationships are the arrows linking the nodes. The result is a directed graph providing to the user an immediate idea about the vulnerabilities and attacks exploiting them, who reported these attacks, and if there is any countermeasure to prevent or mitigate the threats. Moreover, after a first glance giving the idea of the threat scenario, the user can explore the graph to get more information. For each STIX object, a description of the CTI data it represents and external references where the user can find additional information is provided.

The visual representation of CTI information facilitates the process of assessing the severity of a threat. Before diving into security reports, the graph provides a clear view of the threat the user has to deal with and all the aspects related to it. This view can be used by the user as starting point to organize the process of managing the risk derived from discovered threats.

4.4.4 Pentesting Tools

The pentesting tools are a set of instruments to support and automate the penetration testing activities on the SUT. These tools are available for both SAML and OAuth/OIDC and aim to verify whether the SUT suffers any vulnerability. The list of tests is reported in Table 4.3, 4.4 and 4.5, which show the tests targeting any IdM protocol, only SAML and only OAuth/OIDC respectively.

The provided tests can be clustered into three different types: *web application security*, *compliance* and *IdM protocol*. Web application security tests target vulnerabilities common to all web applications. Compliance tests are those tests that verifies if the SUT is compliant with the specifications defined in the IdM protocol standard while IdM protocol tests target vulnerabilities of the IdM protocol implementation that are not covered by compliance tests.

The executed tests can be passive or active. Passive tests analyze the traffic generated during the authentication flow to discover anomalies such as missing parameters required by the standard. Active tests modify the exchanged messages to verify how the SUT reacts to the change. For instance, an active test on an IdM implementation of OAuth/OIDC checks if the `state` parameter used to prevent CSRF attack can be manipulated. During the authentication flow, the `state` parameter is modified

using the proxy APIs. In case the authentication process is completed despite the modification, it means the SUT does not manage correctly the `state` parameter and it might expose the user to CSRF attacks. In case the SUT notices the change, the test fails meaning that the SUT performs adequate verifications on the `state` parameter.

Since passive tests perform only static analysis, they can be executed all at once when the authentication process is completed. However, this is not the case of active tests. Each active test performs some modifications to the messages. The result of these modifications is obtained from the authentication flow outcome, namely whether it is completed or not. Nevertheless, if the tools perform multiple modifications simultaneously, in case of failure the tool can not identify which modification led to the error. For this reason, each active test is executed independently from the others.

To make these tests run automatically, the user has to provide an *authentication trace* to the pentesting tools. The authentication trace is the list of steps the user performs to authenticate on the IdM system (i.e. visit `www.example.com`, click on *Login* button, etc.). This list has to be constructed following the guidelines defined by the tools (more details in Chapter 5). Once the list of steps is inserted, the trace is first tested to ensure it works and then it is used to run the automated checks.

The check results are reported in a table inside the tool. In the table, the user finds a recap with (i) the check performed together with a small description, (ii) whether the test has been successful, and in case of failure, (iii) where the tool detected something not compliant with the standard. The content of this table can be also exported as a file.

To evaluate the completeness of MIG pentesting tools we defined a list of fundamental security checks both for OAuth/OIDC and for SAML. The list is built on top of the work of some colleagues [61, 58] that analyzed vulnerabilities described in the security best practices of SAML, OAuth and OIDC, and vulnerabilities reported in the literature. Once defined the list, we compared the test provided by MIG with the ones performed by the other state-of-the-art tools. For the comparison we considered only those tools performing checks on IdM protocols. Therefore, the comparison does not include general purpose pentesting tools (e.g. BLOCK). We excluded also WS-Attacker because it tests SOAPAction spoofing and WS-Addressing spoofing, two attacks exploiting vulnerabilities of SOAP protocol that is very rare nowadays since actual implementations of SAML usually implement Web Profile based on HTTP.

Given these assumptions, the comparison for test targeting OAuth/OIDC is presented in Table 4.6 while the one with tests targeting SAML is available in Table 4.7. To get a concrete way of comparison between tools, we define a metric we called *coverage*. The coverage represents the percentage of checks performed by a tool out of the total number of checks we identified in the aforementioned list. We think this metric could be a good measure both to compare the pentesting tools and to evaluate their completeness.

Concerning OAuth/OIDC, we noticed that only few tools perform a complete test on the compliance with protocol specification (e.g. the presence of required parameters). Most of the tools focus on a single aspect of compliance (e.g. the `state` parameter presence) or on aspects of the IdM protocol implementation that can lead to particular attacks (e.g. code or access token theft). For instance, there are some aspects such as the `state` parameter or the presence of third-party content that are tested by most of the tools while other tests that are tool-specific. For instance, this is the case of ProFESSOS which sets up a malicious IdP to test the security of a C. Nonetheless, if we look at the overall coverage,

Table 4.3: Collection of tests targeting any IdM protocol.

Test	Target	Type	Description	Mitigation
Use of HTTPS	C, IdP	P	Check if all HTTP messages communicates over a secure channel.	Change web server configuration and forward unsecured HTTP messages to HTTPS.
Clickjacking Prevention	C, IdP	P	Check if all the HTTP messages has the header X-FRAME-OPTIONS set as DENY or SAME-ORIGIN.	Set the header X-FRAME-OPTIONS as DENY or SAME-ORIGIN

P = Passive Test; A = Active Test.

the set of tests provided by MIG pentesting tools can be considered the most complete. The coverage provided by MIG pentesting tools is the highest among the pentesting tools actually available. MIG pentesting tools miss some compliance checks concerning FAPI profile, a profile built on top of OAuth/OIDC used to regulate financial operations. Even if it is important, it is also reasonable to say that it concerns a particular use case of OAuth/OIDC therefore, if we are not working with financial IdM systems, we are not missing a crucial test for the security of our IdM system. The main weakness of MIG pentesting tools is the impossibility to perform tests that simulate a malicious IdP as the one PrOfESSOS performs. This is due to the way MIG pentesting tools are implemented. They are developed as extensions of a proxy and the tests they perform are based on

Table 4.4: Collection of tests targeting IdM systems using SAML

Test	Target	Type	Description	Mitigation
Session hijacking	C	A	Check whether the C does not manage properly the session during the authentication process allowing CSRF attacks	Implement mechanisms to handle properly the user sessions
Relaystate parameter tampering	C, IdP	A	Check if the value of Relaystate parameter can be tampered	Sanitize the value of Relaystate parameter
Presence of ID attribute	C, IdP	P	Check whether the ID element is present in the SAML request	Configure the C to include ID attribute in the SAML request and the IdP to accept only SAMLRequest with ID attribute
Presence of Issuer attribute	C, IdP	P	Check whether the Issuer element is present in the SAML request	Configure the C to include Issuer attribute in the SAML request and the IdP to accept only SAMLRequest with Issuer attribute
SAML request compliant with SPID	C, IdP	P	Check whether the SAML request is compliant with SPID standard	Configure the C to issue SPID compliant SAML request
Presence of Audience element	C, IdP	P	Check whether the Audience element is present in the SAML response	Configure the IdP to include Audience element in the SAML response and the C to accept only SAML response with Audience element
Presence of OneTimeUse attribute	C, IdP	P	Check whether the OneTimeUse attribute is present in the SAML response	Configure the IdP to include OneTimeUse attribute in the SAML response and the C to accept only SAML response with OneTimeUse attribute
Presence of NotOnOrAfter attribute	C, IdP	P	Check whether the NotOnOrAfter attribute is present in the SAML response	Configure the IdP to include NotOnOrAfter attribute in the SAML response and the C to accept only SAML response with NotOnOrAfter attribute
Presence of InResponseTo attribute	C, IdP	P	Check whether the InResponseTo attribute is present in the SAML response	Configure the IdP to include InResponseTo attribute in the SAML response and the C to accept only SAML response with InResponseTo attribute
Presence of Recipient attribute in SubjectConfirmationData element	C, IdP	P	Check whether the Recipient is present in the SAML response	Configure the IdP to include Recipient attribute in SubjectConfirmationData element of the SAML response and the C to accept only SAML response with Recipient attribute
Canonicalization algorithm	C, IdP	P	Check if the Canonicalization algorithm used by the XML parser encodes also comments	Configure IdP and C to use XML parser Canonicalization algorithm that includes comments
SAML response compliant with SPID	C, IdP	P	Check whether the SAML response is compliant with SPID standard	Configure the IdP to issue SPID compliant SAML response

P = Passive Test; A = Active Test.

the analysis of the HTTP traffic intercepted by the proxy (more details in Chapter 5). Given this architecture, it is hard to set up an IdP as PrOfESSOS does. However, we are looking forward to find a way to implements also this type of test in the future.

As for OAuth/OIDC, also in case of SAML, MIG pentesting tools have the highest percentage in terms of coverage. If we look at the type of tests provided, we notice that in case of SAML there is no overlapping between tools but every tool targets a specific class of tests. MIG pentesting tools are the only ones providing protocol compliance verification. In addition, they perform some tests on three other specific vulnerabilities (RelayState integrity [6], session integrity [38] and canonicalization algorithm [51]) leading to severe threats. However, with respect to the other tools, MIG pentesting tools does not implement tests to verify if the target system suffers vulnerabilities leading to XSW and XXE attacks. These checks are difficult to be automated because they require a knowledge of

Table 4.5: Collection of tests targeting IdM systems using OAuth/OIDC.

Test	Target	Type	Description	Mitigation
CSRF Prevention	C, IdP	P	Checks whether state parameter is used	Introduce the state parameter in the authentication flow
Check compliance with the standard	C, IdP	P	Checks whether all the parameters reported as REQUIRED in the standard [4] are present in the considered authentication flow	Introduce the missing parameters in the authentication flow
Adopt <i>PKCE</i>	C, IdP	P	Checks whether <i>Proof Key for Code Exchange (PKCE)</i> extension [60] is implemented	Implement the <i>PKCE</i> extension in the authentication flow
Redirect URI validation	IdP	P	Checks whether redirect_uri value either points to internal domains or accepts values with <code>../</code>	Implement validation on redirect_uri parameter
Check JWT token	C, IdP	P	Checks whether the JWT token is correctly constructed	Insert into the JWT token all the necessary parameters (e.g. <i>iss</i> , <i>aud</i> , etc.)
Token stored as plaintext in cookies	IdP	P	Checks whether the token generated by the IdP is returned as plaintext in the authorization response cookies	Do not return tokens as plaintext in the authorization response cookies
307 HTTP redirect	C, IdP	P	Checks whether 307 HTTP redirect is used for the redirection between C and IdP	Use other redirection mechanisms because this one can lead to user,Ãs credential leakage
Block referer header	C, IdP	P	Checks whether the referrer-policy header of the messages is set to no-referrer	Set referrer-policy of the exchanged messages to no-referrer
External JS files or components	C, IdP	P	Checks whether external Javascript files or components are loaded	Verify that external JS or components loaded do not leak sensitive information, and in that case, remove them
state parameter modification	IdP	A	Change state parameter value in the Authorization Request	Configure the IdP to verify state parameter integrity
state parameter deletion	IdP	A	Delete the state parameter when sent to the IdP with the Authorization Request	Configure the IdP to require the state parameter.
code_challenge parameter deletion	IdP	A	Delete the code_challenge parameter when sent to the IdP with the Authorization Request	Configure the IdP to require the code_challenge parameter.
code_challenge_method parameter modification	IdP	A	Change code_challenge_method parameter value in the Authorization Request	Configure the IdP to verify code_challenge_method parameter integrity
redirect_uri parameter modification	C	A	Change redirect_uri parameter value in the Authorization Request	Configure the C to verify redirect_uri parameter integrity
JWT token modification	IdP	A	Change alg parameter value of JWT token to <i>'null'</i>	Configure the IdP to verify that alg parameter of JWT token is different than <i>'null'</i>

P = Passive Test; A = Active Test.

the structure and the semantic of the XML message, therefore, the process to perform the test is not as simple as checking the presence of an attribute or modifying the value of a parameter. Indeed, EsPreSSO and SAML Raider, the two tools that are capable to test these vulnerabilities, require the intervention of the user that has to inject manually the exploit. Moreover, the outcome of the test is not automatically detected by the tools. Due to their specificity, the user has to analyze the behaviour of the system where the test has been performed and discriminate by himself whether the tested system is vulnerable or not. Therefore, a real automated solution to test these vulnerabilities does not yet exist. However, until this gap is not filled, existing manual solutions are worth to be used. Since both EsPreSSO and SAML Raider work as extensions of a proxy as MIG pentesting tools do, the upload feature provided by MIG can be used to add them to the available pentesting tools, so that also vulnerabilities leading to XSW and XXE can be tested with MIG. Doing this, if we compute a new coverage combining the checks of MIG pentesting tools with the ones of EsPreSSO and SAML Raider, we are able to reach a complete coverage of 100%.

Table 4.6: OAuth/OIDC pentesting tools comparison

Test	FAPI conformance suite [30]	Impersonator [10]	OAuch [26]	OAuth Detector [24]	OAuthGuard [13]	OAuthTester [25]	OpenID Attacker [28]	OVERSCAN [12]	PrOESSOS [29]	SSOScan [11]	WPSE [38]	MIG Pentesting tools
Web Application Security												
Secure headers [62]	-	-	-	-	-	-	-	✓	-	-	-	✓
Use of HTTPS [63]	-	-	-	-	✓	-	-	✓	-	-	-	✓
Compliance												
Block referrer header [56]	✓	-	✓	-	-	-	-	-	-	✓	✓	✓
<code>code</code> , <code>access_token</code> or <code>id_token</code> leakage via third party content [56]	✓	-	✓	-	✓	-	-	-	-	✓	-	✓
FAPI [64]	✓	-	-	-	-	-	-	-	-	-	-	-
<code>redirect_uri</code> integrity [56]	✓	-	✓	-	-	-	-	-	-	-	✓	✓
<code>redirect_uri</code> modification with <code>../</code> [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
<code>redirect_uri</code> without subdomain [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
Require <code>code_challenge</code> [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
Require <code>code_challenge_method</code> [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
<code>state</code> parameter presence [56]	✓	-	✓	✓	✓	✓	-	✓	-	-	✓	✓
<code>state</code> parameter modification [56]	✓	-	✓	-	-	✓	-	-	-	-	-	✓
Token saved as Cookies [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
Use of PKCE [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
Validated JWT [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
Well formed JWT [56]	✓	-	✓	-	-	-	-	-	-	-	-	✓
307 redirect attack [56]	✓	-	✓	-	-	-	-	-	-	-	✓	✓
IdM protocol												
Access token misuse [11]	-	-	-	-	✓	-	-	-	✓	✓	✓	-
Signed request misuse [38]	-	-	-	-	-	-	-	-	-	✓	-	-
Replay attack [29]	-	✓	-	-	-	-	-	-	✓	-	-	-
<code>state</code> parameter replay [25]	-	-	-	-	-	✓	-	-	-	-	-	-
<code>state</code> not bind to the user [25]	-	-	-	-	-	✓	-	-	-	-	-	✓
Token theft via malicious IdP [29]	-	-	-	-	-	-	✓	-	✓	-	✓	-
Coverage	65%	4%	61%	4%	17%	17%	4%	13%	9%	17%	26%	74%

✓ = provided check; - = check not provided

Table 4.7: SAML pentesting tools comparison

Test	EsPReSSO [34]	SAMLyze [37]	SAML Raider [33]	WPSE [38]	MIG Pentesting tools
Web Application Security					
Secure headers [62]	-	-	-	-	✓
Use of HTTPS [63]	-	-	-	-	✓
Compliance					
Presence of Audience element [59, 58]	-	-	-	-	✓
Presence of ID attribute [59, 58]	-	-	-	-	✓
Presence of InResponseTo attribute [59, 58]	-	-	-	-	✓
Presence of Issuer attribute [59, 58]	-	-	-	-	✓
Presence of NotOnOrAfter attribute [59, 58]	-	-	-	-	✓
Presence of OneTimeUse attribute [59, 58]	-	-	-	-	✓
Presence of Recipient attribute [59, 58]	-	-	-	-	✓
Presence of Signature [59, 58]	-	-	-	-	✓
SAMLRequest compliant with SPID [65]	-	-	-	-	✓
SAMLResponse compliant with SPID [65]	-	-	-	-	✓
IdM protocol					
Canonicalization algorithm [51]	-	-	-	-	✓
RelayState integrity [6]	-	-	-	-	✓
Session integrity [38]	-	-	-	✓	✓
XML Entity Expansion [36, 66]	✓	✓	-	-	-
XML External Entity [36, 66]	✓	✓	-	-	-
XML Signature Exclusion [35]	-	-	✓	-	-
XML Signature Wrapping [35]	-	-	✓	-	-
Coverage	10%	10%	10%	5%	79%

✓ = provided check; - = check not provided

5 Implementation

The implementation of MIG architecture has been an incremental process. We started from C and IdP instances that have been configured to run in a sandbox. After that, MSC Logger running on the proxy and the complementary MSC Drawer webapp have been developed. STIX visualizer has been then integrated with MSC Drawer. To develop the pentesting tools, after a research of the available options, we decided to develop proxy extensions performing automated checks using Selenium. Finally, the dashboard to set up an IdM system in a sandbox using MIG Backend instances and to configure MIG Frontend tools completed MIG architecture.

The following sections present the implementation details of each component.

5.1 Backend

5.1.1 Client and Identity Provider Repositories

The IdM instances presented in Section 4.2.1 have been selected among the most famous implementation recognized nowadays [48, 49, 50]. The selected instances were firstly configured and deployed in a local environment. Several tests were performed to get a deep understanding of the configuration of the instances. For instance, if we take as example an IdM system based on SAML, we firstly deployed C and IdP. Then we investigated how to change the ports where they run. This is important because we want flexible instances that adapt to the environment where they are deployed therefore we have to allow the user to customize the ports where these services will run. After that, we looked at metadata, an XML-document describing the characteristics of a SAML entity (i.e. certificates, URL endpoints, etc.). This document is necessary because it is exchanged between C and IdP during the federation process to establish a trusted relationship between the two. We aimed to make this metadata static so that once they are generated the first time, then they can be reused without the need to perform the federation every time. However, if we modify the ports where SAML entities run, we have also to change the respective metadata file. These changes are performed through a find-replace process where we substitute standard ports with the customized ones (i.e. `sed` command in Listing 5.1). Doing this, we notice that metadata is not invalidated therefore we achieved our purpose of performing federation the first time and then reuse the same metadata every subsequent time a SAML IdM system is deployed.

Listing 5.1: Example of the Dockerfile of a C implementing SAML

```
# load base image
FROM ubuntu:16.04

# initialize customized configuration parameters
ARG c_version
ARG c_port
ARG idp_port
ENV C_PORT=$c_port
ENV IDP_PORT=$idp_port

# install necessary packages
RUN apt-get update
RUN apt-get install -y default-jdk nano unzip curl

# copy chosen C implementation
```

```

COPY src/spring-security-saml-sp-1.0-$c_version.jar /spring-security-saml-sp-1.0.jar

# add metadata
RUN mkdir /metadata
COPY src/idp-metadata.xml /metadata

# customize metadata
RUN sed -i 's+localhost/idp/profile+localhost:'$IDP_PORT'/idp/profile+g' /metadata/
    idp-metadata.xml;

# deploy and run the C
CMD java -jar spring-security-saml-sp-1.0.jar --server.port=$C_PORT

```

Each deployment was then moved to a Docker container. The container is generated from a Dockerfile where the steps for the initialization, the configuration and deployment are listed. Listing 5.1 represents an example of the Dockerfile of a C implementing SAML. In every Dockerfile, two important variables are set: `C_PORT` and `IDP_PORT`. These variables will contain the values of the ports where C and IdP will run. Both are initialized using some arguments passed to the Dockerfile at the moment of the creation of the container. These arguments are those preceded by the keyword `ARG` in Listing 5.1. As we can see in the latter code snapshot, in addition to C and IdP ports, a third argument is passed to the Dockerfile. This argument, `c_version`, is used in some cases to tell the container which version of the C has to be loaded. However, on the contrary of `c_port` and `idp_port`, it is not always present. A similar configuration process has been also applied to OAuth/OIDC instances.

5.1.2 STIX Repository

CTI information provided in MIG has been mapped semantically to STIX objects following the directives provided by STIX specification [67]. The four types of STIX objects we used are:

- *Identity*: this type of STIX object represents actual individuals, organizations, or groups. For instance the university can be classified as Identity.
- *Vulnerability*: this type of STIX object describes a weakness or defect in the requirements, designs, or implementations of the computational logic (e.g., code) found in software and some hardware components (e.g., firmware) that can be directly exploited to negatively impact the confidentiality, integrity, or availability of that system.
- *Course of action*: this type of STIX object describes an action taken either to prevent an attack or to respond to an attack that is in progress. It may describe technical, automatable responses (applying patches, reconfiguring firewalls) but can also describe higher level actions like employee training or policy changes. For example, a course of action to mitigate a vulnerability could describe applying the patch that fixes it.
- *Attack pattern*: this type of STIX object describes ways that adversaries attempt to compromise targets. Attack Patterns are used to help categorize attacks, generalize specific attacks to the patterns that they follow, and provide detailed information about how attacks are performed. An example of an attack pattern is “spear phishing”: a common type of attack where an attacker sends a carefully crafted e-mail message to a party with the intent of getting them to click a link or open an attachment to deliver malware.

Vulnerabilities have been mapped to *Vulnerability* object type, threats to *Attack pattern* type and mitigations to *Course of action*. For each object, name, description and external references where additional information can be found have been specified. In case of a vulnerability we add also few attributes describing the class previously defined (Section 4.2.2) while in case of threats we add information about the risk level. An example of a vulnerability encoded in STIX format is reported in Listing 5.2.

Listing 5.2: Example of STIX JSON Object

```
{
  "type": "vulnerability",
  "id": "vulnerability--cc39a329-ee9f-428c-a5a3-d48659c343dd",
  "created": "2020-02-27T14:41:29.544Z",
  "modified": "2020-02-27T14:41:29.544Z",
  "name": "XML External Entity (XXE) processing",
  "description": "XML Parser allows XML External Entities inside SAML Messages and is
    wrongly configured to enable large XML document consumption. ",
  "labels": ["Web Browser SAML/SSO Profile", "SAMLRequest"],
  "external_references": [{
    "source_name": "XXE Cheat Sheet",
    "url": "https://web-in-security.blogspot.it/2016/03/xxe-cheat-sheet.html"
  }], {
    "source_name": "Owasp XML External Entity",
    "url": "https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Processing"
  }],
  "x_FBK_vulnerabilityClass": "Missing XML Validation Class",
  "x_FBK_vulnClass_description": "All the vulnerabilities regarding a missing or an
    improper validation of the XML message that may hide malicious content",
  "x_FBK_provider": "IdP"
}
```

5.2 Dashboard

The dashboard is meant as the component used to create a local IdM system running in a sandbox and to configure MIG Frontend tools. It is a web application developed in NodeJS¹ that provides two main pages: the form where all the components are configured and a page with the instructions to setup MIG environment.

The form containing the fields for the configuration (Figure 5.1) allows the user to customize the following parameters:

- **Proxy port:** set the port where the proxy will run
- **MSC Drawer port:** set the port where MSC Drawer will run
- **STIX Visualizer port:** set the port where the STIX Visualizer will run
- **Target:** specify whether the user is going to test an IdM system in a sandbox or a real system in the wild
- **Protocol:** specified which IdM protocol will be analyzed. The selection of the protocol is important because it is used by the dashboard to automatically select the adequate pentesting tools
- **Scenario name:** an identifier for the scenario the user is going to test. The string inserted in this field will be the `sessionId` the user has to use to access the message sequence chart in MSC Drawer webapp

In case the user selects as target the IdM system in a sandbox, he has also to specify whether he wants to test an uploaded IdM system or create one using the instances available in MIG. In the latter case, the user has to provide the following information:

- **Identity Provider:** select an IdP from the list of the available instances
- **IdP port:** set the port where the IdP will run

¹<https://nodejs.org/>

Dashboard

MIG Frontend configuration

Proxy port: 8080

MSC Drawer port: 5000

STIX Visualizer port: 5555

Upload additional tool

Target:
☐ Wild
☒ Sandbox

Protocol: SAML

Scenario name: saml-example

☒ Create IdM
☐ Upload IdM

IdM system configuration

IdP version: Shibboleth 3.1.1

Idp port: 9443

Client version: Spring SP Base

Client port: 8001

XSS
CSRF

Username: user

Password: password

Download

Figure 5.1: Dashboard architecture

- **Client:** select a C from the list of the available instances
- **C port:** set the port where the C will run
- **Tester credentials:** insert customized credentials (username and password). The inserted credentials will be automatically added to the ones available to authenticate at the selected IdP

Alternatively, in case the user chooses to upload its own IdM system, he has to provide an IdM system containing an already configured couple of IdP and C. To be sure that the uploaded IdM system is compliant with MIG, we have defined a list of guidelines that have to be matched:

- both C and IdP instances must be configured to work on Docker
- the deployment of the imported IdM system must be configured and executed through a `docker-compose` file. The structure of the `docker-compose` file has to follow the example reported in Listing 5.3
- the uploaded IdM system must have at least two containers called *client* and *identityprovider*
- the IdM system must be uploaded as `zip` folder containing the `docker-compose` file in its root folder

Listing 5.3: Example of the `docker-compose` file structure for uploaded IdM system

```

version: '3'
services:
  identityprovider:
    container_name: idp_server
    build:
    context: ./idp/path-to-idp-files-folder
    ports:
    - 9000:9000
  client:

```

```

container_name: client_server
build:
context: ./client/path-to-client-files-folder
ports:
- 8888:8888

```

As for the IdM systems, the user can also upload pentesting tools developed by third parties. The uploaded tools have to be compliant with the MIG testing environment, therefore we have defined a list of guidelines that has to be matched:

- the extension has to be developed as Burp extension
- the extension has to be a `jar` file
- the extension has to specify whether it is going to perform tests on a particular protocol or it is meant to be used for general purpose tests

Once the user inserted the required information and eventually uploaded custom IdM systems or pentesting tools, the submission of the form starts the automatic generation of the needed files. A folder with the “*Scenario name*” inserted is created. Into this folder the files necessary to run MIG Frontend tools are copied and, in case the target is a IdM system in a sandbox, the files necessary for its deployment are also copied. Finally, the files to automatically configure and execute all the components are customized using the inserted information.

Basically there are five configuration files. Three of these are used to configure the proxy and the MSC Logger extension. The fourth is a `docker-compose` file to configure and deploy the MSC Drawer and STIX Visualizer webapps and the last is another `docker-compose` file used in case the target of the pentesting activities is an IdM system running in a sandbox. This last file is not present in case the target of the analysis is a real system in the wild while it is provided by the user in case the IdM system is uploaded (Listing 5.3).

In case the user has chosen to test an IdM system created from the instances provided by MIG, the `docker-compose` file to automatically configure and execute the IdM system is structured as follow:

Listing 5.4: Example of the `docker-compose` file for the IdM system

```

version: '3'
services:
identityprovider:
container_name: idp_server
build:
context: ./idp/path-to-idp-files-folder
args:
- idp_port=9000
- c_port=8888
ports:
- 9000:443

credential:
container_name: credential_db
build:
context: ./credential/path-to-credential-database-folder
args:
- testerUsername=mig
- testerPassword=mig

client:
container_name: client_server
build:

```

```

context: ./client/path-to-client-files-folder
args:
- idp_port=9000
- c_port=8888
ports:
- 8888:8888

```

The `docker-compose` file in Listing 5.4 describes three services that will be deployed in three different containers: *identityprovider*, *credential* and *client*. Each of them has some common properties. `container_name` that represents the label used by Docker to identify the container. `build` with two sub-properties, `context` and `args`, describing respectively where the files necessary to build the container service are located and the values of the parameters that will be passed to the Dockerfile at the moment of the container creation (as described in Section 5.1.1). `ports` attribute represents instead the mapping between the port of the local host (the one on the left of the colon) and the port of the container (the one on the right of the colon). For instance, in case of *identityprovider* container in Listing 5.4, the container port 443 can be accessed from the local machine at port 9000.

However, while *identityprovider* and *client* represent the selected instances of IdP and C, *credential* is never mentioned before. This service is added to the `docker-compose` to allow the customization of the credentials the user will use to authenticate at IdP. Indeed, the container receives as argument username and password inserted by the user in the dashboard. The `docker-compose` file in Listing 5.4 represents the minimal configuration for the IdM system we provide in MIG. Depending on the IdP/C instance the user selects, more arguments can be passed to the respective containers.

The `docker-compose` file used to configure and run MSC Drawer and STIX Visualizer (Listing 5.5) is similar to the previous one (Listing 5.4). Both MSC Drawer and STIX Visualizer webapps have been configured to run on Docker and a container for each one has been created. Additionally, a third container where MongoDB instance runs has been added. This container hosting MongoDB is used by MSC Drawer to store the information about the messages that it will draw. As for the `docker-compose` used to configure and run the IdM system, also in this case we use arguments passed to MSC Drawer and STIX Visualizer to configure the ports where the services will run.

Listing 5.5: Example of the `docker-compose` file for MSC Drawer and STIX Visualizer webapps

```

version: '3'
services:
mscdrawer:
container_name: msc-webapp
build:
context: ./path-to-msc-webapp-files-folder
args:
- msc_port=5000
- stix_port=5555
ports:
- 5000:5000
links:
- mongo

mongo:
container_name: mongo-msc
image: mongo

stixvisualizer:
container_name: stix-visualizer
build:
context: ./path-to-stix-visualizer-files-folder
args:

```

Figure 5.2: Interface to add a new filter

```
- stix_port=5555
ports:
- 5555:5555
links:
- mongo
```

The files used to configure the proxy and MSC Logger extension are three JSON files called **project-options**, **user-options** and **msc-logger-options**. The first two are used to automatically set up the proxy while the last is used for MSC Logger. **project-options** file is used to configure the port where Burp proxy will run. **user-options** contains instead the list of the extension that has to be loaded. In case the chosen protocol is SAML, this file is used to load only SAML pentesting tool and vice-versa in the case of OAuth/OIDC. The **msc-logger-options** file used to configure MSC logger filters and other properties is described in details in Section 5.3.1.

All these configuration files are customized in background through a find-replace process. Targeted placeholders are substituted with the information inserted by the user. Once the replacement process is finished, the steps to run MIG together with the location of the files folder are displayed in the dashboard webapp. The same information is also written in a **README** file added to the aforementioned folder.

5.3 Frontend

5.3.1 Proxy

The proxy we decided to adopt is Burp. This choice has been done because it runs on every operating system and it provides a set of useful and easy-to-use APIs for the development of the pentesting tools. MSC Logger, the extension of the proxy used to communicate with the MSC Drawer, exploits these APIs to implement the *Intercept*, *Filter* and *Parse* features described in Section 4.4.1.

When a message is intercepted by the proxy, the method **processHttpRequest** is called. This method receives as input the intercepted raw HTTP message which is inspected and manipulated MSC Logger. It then forwards the raw data to the filter we implemented. Following a white-list approach, the filter uses the conditions inserted by the user to decide whether the intercepted message has to be drawn in the MSC Drawer. The conditions are specified using the input form depicted in Figure 5.2. Burp manages messages as couples of request and response therefore the user can specify conditions for the following fields: hostname, request headers, request parameters, response headers and response body. For instance, by writing *google* into the hostname field, all the messages containing *google* inside the hostname will be drawn in the MSC drawer.

In case the intercepted message satisfies the filters, before pushing the data to the MSC Drawer, the raw information has to be parsed and formatted in the proper way. In case some parameters are exchanged encoded (i.e. **SAMLRequest** and **SAMLResponse**), MSC logger offers the possibility to decode them using an external service. Additionally, there is also the possibility to define aliases to improve the clarity of the message sequence chart. For instance, the user can substitute the name of the IdP with a custom one that helps him to better understand the represented flow. Once the selected parameters have been decoded and the aliases applied, MSC Logger formats the raw message and the decoded data according to the API specification of MSC Drawer (see more in Section 5.3.2) and pushes it to the MSC Drawer webapp.

To easily reuse the inserted configurations such as filters and aliases, MSC Logger allows the exportation

of this configuration in a JSON² file that can be imported the next time the user will set up the proxy in the same way. The JSON configuration file can be used to configure all the aspects of the extension therefore the user could also write a customized file and then import it. To be interpreted by MSC logger, the configuration file must have the following parameters:

- **host**: hostname where the MSC Drawer is running
- **port**: port where the MSC Drawer is running
- **protocol**: protocol supported by the MSC Drawer. It can be HTTP or HTTPS
- **sessionID**: string used to identify the MSC Drawer session where intercepted messages are represented in the message sequence chart (see Section 5.3.2 for more details)
- **readonly**: property that could be used to make the drawn messages persistent. It is thought to share the message sequence chart among multiple users without the possibility to modify it. In that case, the value has to be set to **true**. In other cases, such as MIG scenarios, the property value has to be **false**
- **filterItems**: this field contains an array of JSON Object describing the filters. Each JSON Object represents a filter entry and it has to be structured as follow:
 - *host*: list of string separated by ; representing the values that have to be searched in the host of the intercepted message
 - *request_header*: list of string separated by ; representing the values that have to be searched in the headers of the intercepted request
 - *request_params*: list of string separated by ; representing the values that have to be searched in the parameters of the intercepted request
 - *response_header*: list of string separated by ; representing the values that have to be searched in the headers of the intercepted response
 - *response_body*: list of string separated by ; representing the values that have to be searched in the body of the intercepted response
 - *draw*: represent which messages have to be drawn by the MSC Drawer. 0 stands for both request and response, 1 for only the request while 2 for only the response

The ; represents the OR conjunction. For instance, if the field *host* contains the value “*google; facebook*”, MSC Logger will look for messages containing in the host field either the value “*google*” or “*facebook*”. On the contrary, fields of the same JSON Object are evaluated jointly, as if there is an AND conjunction between them. This means that if we add to the previous filter the string “*token*” into the **request_param** field, MSC Logger will match all the messages containing the value “*google*” or “*facebook*” in the host field and simultaneously having a parameter of the request containing the value “*token*”. Finally, each JSON Object is evaluated independently therefore each message is tested against all the filters and in case it matches one of them, it is drawn in the MSC Drawer webapp.

- **aliasItems**: this field contains an array of JSON Object describing the aliases that can be applied to the intercepted messages to improve the clarity of the MSC that will be drawn. Using this feature, the user might decide to rename the hostname of the C with a custom name he decides. For instance, in an IdM system using SAML the C could be renamed as SP. To do that, each alias entry has to be a JSON Object structured as follow:

- *target*: the string that has to be replaced (i.e. C)
- *alias*: the string that replaces the target value (i.e. SP)

²<https://www.json.org/json-en.html>

- **parserItems**: this field defines the external services used to decode encoded parameters. It contains an array of JSON Object, each one structured as follow:
 - *encoding*: identifier of the encoding format the service is able to decode (e.g. **base64**)
 - *encodingURL*: URL of the API endpoint providing the decoding service (e.g. **www.example.com/decode**)
 - *method*: the method used to call the API endpoint providing the service. It can be either GET or POST
 - *paramName*: name of the parameter where the API service expects the string that has to be decoded (e.g. **inputValue**)

For instance, if we consider the values *encoding* equals to **base64**, *encodingURL* equals to **www.example.com/decode**, *paramName* equals to **inputValue** and GET as HTTP method, we are telling MSC Logger that **base64** string can be decoded by calling the API endpoint in the following manner: **GET www.example.com/decode?inputValue=<StringToBeDecoded>**

- **parserMappingItems**: this field contains an array of JSON Object describing the parameters that have to be decoded and the decoded format that has to be used. For instance, in an IdM system using SAML, we might find the mapping between the parameter **SAMLRequest** and the encoding format **base64**.

Each JSON Object describing a mapping has to be structured as follow:

- *param*: name of the parameter that has to be decoded (i.e. **SAMLRequest**)
- *encoding*: encoding format that has to be used (i.e. **base64**). Note that this field takes the value from the list of encoding formats defined in **parserItems** field

The configuration file created can be imported when the extension is loaded. For instance, suppose that MIG is used for educational purposes at the university. The professor wants to show in practice to students how the OAuth/OIDC protocol flow works. To do that, the professor can set the filters needed to draw the authentication message sequence chart of OAuth/OIDC, export the configuration and share it with the students. The latter, just by importing the file, will be able to draw the corresponding message sequence chart. Therefore, we can see that with few operations an entire class of students without the knowledge necessary to use the MSC Logger will be able to exploit its features. Moreover, to improve the automatic configuration of MSC Logger, we also add the possibility to automatically import the configuration when the extension is loaded on Burp. At the startup, MSC Logger looks for a file called **msc-logger-options.json**. If it finds the file, the contained configuration is loaded.

Listing 5.6: MSC Logger configuration used to draw OAuth/OIDC message sequence chart

```
{
  "host": "localhost",
  "port": "5000",
  "protocol": "HTTP",
  "sessionId": "test-logger",
  "readonly": false,
  "filterItems":
    [{
      "response_header": "",
      "request_header": "",
      "host": "",
      "request_params": "redirect_uri; code;",
      "response_body": "",
      "draw": 0
    }],{
```

```

    "response_header": "",
    "request_header": "",
    "host": "",
    "request_params": "",
    "response_body": "code;",
    "draw": 0
  }],
  "aliasItems": [
    {
      "target": "localhost:8888",
      "alias": "Authorization Server"
    }
  ],
  "parserItems": [],
  "parserMappingItems": []
}

```

Nonetheless, until the user does not get used to MSC Logger, defining its own configuration both via user interface and JSON file can be really tricky. Therefore, to further simplify the process of drawing the message sequence chart of IdM protocol, together with MSC Logger we provide two configuration files customized to draw OAuth/OIDC and SAML authentication flow. An example of the OAuth/OIDC one is presented in Listing 5.6.

5.3.2 MSC Drawer

The MSC drawer is a webapp developed in NodeJS responsible for the representation of the message sequence chart of the flow intercepted by the proxy extension MSC Logger. The webapp can display multiple message sequence chart simultaneously therefore it is organized in sessions. Each session can be accessed through a `sessionId` defined by the user. All the messages added to the MSC Drawer webapp are stored in a database based on MongoDB platform³. Since we do not have to perform any joins between tables but instead we have lot of objects structured in the same manner, we decided to use this non-relational database that manages data using JSON-like documents. It is really integrated with NodeJS and provides a fast and complete solution that fits perfectly with the needs of MSC Drawer. To interact with the database, the webapp exposes two APIs: one to add a new message to the message sequence chart of a particular session and the other to delete all messages displayed in a session.

The API to add a new message to a session has to be called with the following parameters:

- **Method:** POST
- **URL:** `http://<MSC_DRAWER_HOST>/api/message`
- **Body parameters:**
 - *sessionId*: the identifier of the session where the message will be displayed
 - *from*: the sender entity
 - *to*: the receiver entity
 - *label*: the text that will be displayed on the arrow
 - *headers*: the JSON array with a JSON Object for each header of the message. Each JSON Object has to be formatted as follow:
 - * *name*: the name of the header
 - * *value*: the value of the header
 - *params*: the JSON array with a JSON object for each parameter of the message. The JSON object has to be formatted as follow:

³<https://www.mongodb.com/>

- * *name*: the name of the parameter
- * *type*: the type of the parameter. It can be URL, Body or Cookie
- * *value*: the value of the parameter
- *body*: the body of the message

An example of a call to the API to add a new message is reported in Listing 5.7.

Listing 5.7: Example of the addition of a new message to the session `test`

```
Method: POST
URL: http://localhost:5000/api/message
Body:
{
  sessionId: test,
  from: fbk.eu,
  to: User Agent,
  label: 1. HTTP/1.1 302 fbk.eu/idp/profile/SAML2/Redirect/SSO,
  headers:
  [{
    name: HTTP/1.1,
    value: 302 Found
  },
  {
    name: Date,
    value: Tue, 04 Feb 2020 15:30:17 GMT
  }],
  params:
  [{
    name: RelayState,
    type: URL,
    value: cookie%3efrw1580830217_305f
  },
  {
    name: SAMLRequest,
    type: URL,
    value: <samlp:AuthnRequest>...</samlp:AuthnRequest>
  }],
  body: ""
}
```

The API to delete all messages of a session has to be called with the following parameters:

- **Method:** DELETE
- **URL:** `http://<MSC_DRAWER_HOST>/api/message/<sessionId>`

For instance, all the messages of the session `test` can be deleted with the API call displayed in Listing 5.8.

Listing 5.8: Example of the deletion of all messages of the session `test`

```
Method: DELETE
URL: http://localhost:5000/api/message/test
```

Both APIs are automatically called by the MSC Logger extension of Burp, therefore the user does not need to push or delete any message manually. To access the message sequence chart where the traffic intercepted by Burp is displayed, the user has to visit the MSC Drawer webapp and login using his `sessionId` (see Figure 5.3). The `sessionId` is chosen by the user during the configuration of MIG



The diagram illustrates the SAML authentication process involving three main components: the User Agent, the Service Provider (esse3.unitn.it), and the Identity Provider (idp.unitn.it).

Search Interface: At the top right, there is a search bar with the text "Search:" and a search button. Below it, there are three checkboxes: "URL" (checked), "Headers" (checked), and "Parameters" (checked).

Sequence of Operations:

- 1. GET esse3.unitn.it/auth/Logon.do**: The User Agent sends a GET request to the Service Provider.
- 1. HTTP/1.1 302 idp.unitn.it/idp/profile/SAML2/Redirect/SSO?RelayState,SAMLRequest**: The Service Provider responds with a 302 redirect to the Identity Provider.
- 2. GET idp.unitn.it/idp/profile**: The User Agent sends a GET request to the Identity Provider.
- 3. [Request to IDP]**: The User Agent sends a request to the Identity Provider (partially obscured).
- 4. POST esse3.unitn.it/Shibboleth.sso/SAML2/Redirect**: The Identity Provider sends a POST request back to the Service Provider.
- 4. HTTP/1.1 302 esse3.unitn.it**: The Service Provider responds with a 302 redirect back to itself.

Parameters: A table showing the parameters of the POST request from the IDP to the SP:

Type	Name	Value
URL	RelayState	cookie%3A1590763938_ace
URL	SAMLRequest	I2LNsbsIwEIRUfJi9%2F6Efi0sicCgSLRGhPRHSocmmsRTs1OuU9u3rJfDohaPI2ZmdTzHdmhauh0LXbw2QFq6z7QcFvQCKTDROw6...

Body: A snippet of the SAML response body is shown, starting with the XML declaration and the root element:

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<html>
<head>
<title>302 Found</title>
</head>
<body>
<h1>Found</h1>
<p>The document has moved <a href="http://idp.unitn.it/idp/profile/SAML2/Redirect/SSO?SAMLRequest=I2LNsbsIwEIRUfJi9%2F6Efi0sicCgSLRGhPRHSocmmsRTs1OuU9u3rJfDohaPI2ZmdTzHdmhauh0LXbw2QFq6z7QcFvQCKTDROw6...">here</a>
</p>
</body>
</html>
```

using the dashboard (more details in Section 5.2). Once accessed his session, the user sees in real-time every new message that is added to the message sequence chart (Figure 5.4). The message sequence chart is drawn using JS Sequence Diagrams library⁴. This library takes as input simple strings properly formatted and converts them into figures. Each time a new message is added, the webapp refreshes the displayed message sequence chart.

5.3.3 STIX Visualizer

However, if the user wants to access CTI information about the whole authentication flow performed

⁵<https://github.com/oasis-open/cti-stix-visualization>



Type	Name (click  for CTI information)
URL	RelayState 

Figure 5.5: Button to look up CTI information

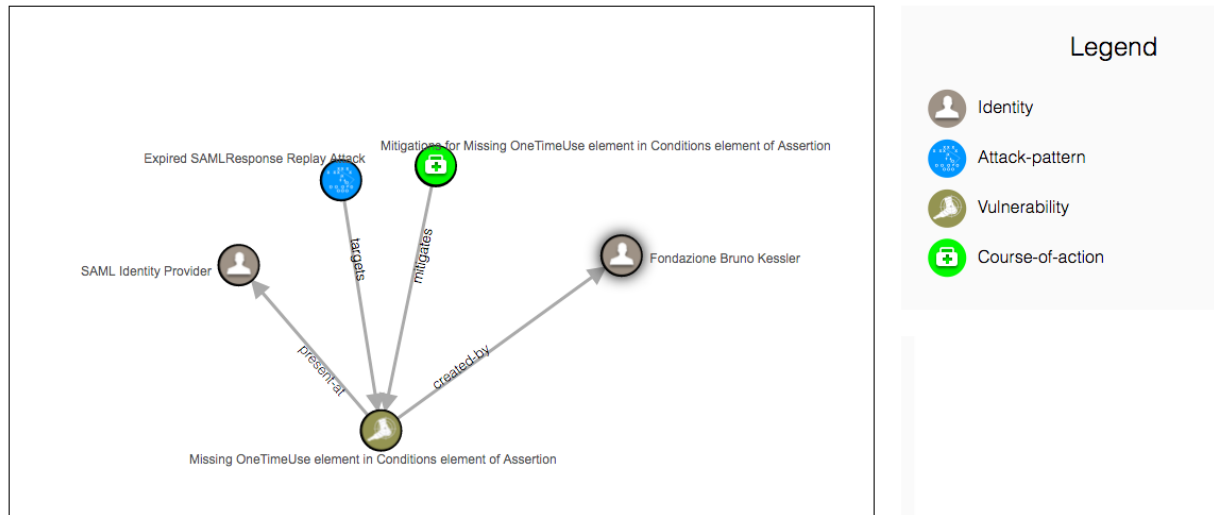


Figure 5.6: STIX Visualizer search result

or a particular entity involved he can use the three red button on top of Figure 5.4. These buttons look up CTI information at different peculiarity levels:

- **Complete:** CTI information retrieved is related to all the messages displayed
- **Host:** CTI information retrieved is related to the messages generated from a host involved in the authentication flow (i.e. `idp.unitn.it` host in Figure 5.4)
- **Custom:** custom CTI information based on the string the user provides as input is retrieved. This can be considered a custom search of CTI information

5.3.4 Pentesting Tools

The two pentesting tools we provide, one for SAML and one for OAuth/OIDC, are developed as extension of Burp proxy. This choice has been done because we can exploit the APIs provided by Burp to easily inspect and manipulate the exchanged messages. As previously said, the provided pentesting tools perform automated tests starting from a trace. This trace describes the steps the user does to authenticate on the SUT. The steps are described in terms of simple interactions with the browser such as click on “Login” button and type “bob” in the field with ID equals to “username”. To execute the steps we exploit Selenium, the most used browser automation library for testing web-applications. Selenium automatically simulates a web browser and the interactions the user performs on it. We provide passive and active tests. Passive tests analyze the HTTP traffic generated during the authentication flow to discover anomalies while active ones perform some modification to the exchanged messages to see how the system reacts to the change. If passive test results are obtained at the end of the authentication flow analysis, active test results depend on the outcome of the modified authentication flow. For this reason, each active test needs to be executed isolated from the others. Indeed, in case we perform multiple modifications during the same authentication flow and an error is generated, we can not infer which modification has generated that error. Therefore, given a trace properly formatted, we use Selenium to automate the authentication process and replicate it as many times as we need. This aspect is crucial for the implementation of our pentesting tools.

The trace used to automate the authentication flow accepts the following commands:



Figure 5.7: Test results example

- **open** | **http://www.example.com**: open web driver browser at the indicated URL (`http://www.example.com` in this case)
- **type** | **indicator=example** | **text**: type the string “text” at the location identified by the indicator
- **click** | **indicator=example**: click element of the page identified by the indicator

where *indicator* can be:

- *name*: name of the HTML element Selenium will look for
- *class*: class of the HTML Selenium will look for
- *id*: id of the HTML element Selenium will look for
- *xpath*: xpath of the HTML element Selenium will look for
- *link*: link inside the HTML page

For instance, when Selenium executes the trace in Listing 5.9, it opens the browser and visits `http://www.example.com`. Then it types inside the HTML element which name is “username” the string “jford” and clicks on the button having id equals to “nextBtn”.

Listing 5.9: Example of Selenium trace

```
open | http://www.example.com
type | name=username | jford
click | id=nextBtn
```

However, these steps do not necessarily have to be written manually. The user can exploit Katalon⁶. Katalon is web browser plugin that allows the automatic registration of the actions performed in the browser and the exportation in several formats, including the one compatible with our pentesting tools. Therefore, to run the automated test the user has to perform the authentication on the SUT, record this process using Katalon plugin and provide the generated trace to the pentesting tools. Once inserted the trace in the tool, the tester has to choose the browser he wants to use to execute the automated tests. Google Chrome and Mozilla Firefox are actually the two browsers supported by Selenium. Based on the selected browser, the user has to import the respective webdriver binary file that can be downloaded from Selenium website. This file is used by Selenium to execute the automated operations on the browser. The last step before running the automated checks is the test of the trace. The pentesting tool tries to execute the inserted trace. In case of successful execution, then the user can start running the automated tests, otherwise, the trace has to be fixed and checked again.

If everything works, the automated tests are executed. Concerning passive tests, the testing strategy is quite simple: the intercepted authentication flow is compared to the expected ones and, in case of anomalies, these are reported to the user. For instance, the pentesting tool analyzes the HTTP traffic checking if PKCE [60] is implemented. In case this protection mechanism is implemented, the tool will color the test in green. Otherwise, it will color the test in red and report the discovered anomaly to the user. Figure 5.7 shows an example. The same highlighting method is used to report the results of active tests but the testing strategy applied is different. If the trace is correctly executed despite the modification performed to the authentication flow, the SUT is considered as vulnerable. On the contrary, if the trace execution is not completed for some reasons (e.g. an error is generated), the SUT is considered secure with respect to the tested vulnerability. At the end of the execution of the automated tests, the results are reported in a table that can be also exported as JSON file.

⁶<https://www.katalon.com/>

5.4 Usage

The ideal sequence of steps to use MIG are the following:

1. clone the project repository available at <https://github.com/stfbk/micro-id-gym>
2. follow the instruction provided in the repository to run the dashboard
3. configure MIG Frontend tools
4. select the target. Here the user has to choose whether he wants to test a real scenario in the wild or a sandboxed one locally. Based on his choice, he moves forward to step 6, in case of real system, or step 5 in case of local system.
5.
 - (a) in case the user decides to recreate an IdM system locally, he has to pick the C and the IdP from the instances provided by MIG
 - (b) in case the user decides to upload his customized IdM system, he has to upload it through the dashboard
6. download the files for the MIG Frontend tools and, in case of a sandboxed system, also the files to deploy the IdM system locally
7. run MIG Frontend tools and, in case of a sandboxed system, deploy the IdM system that hereafter is referred as SUT
8. perform the authentication on SUT. The messages exchanged during this process are automatically displayed in the MSC Drawer.
9. look at the created message sequence chart to verify if it is compliant with the one described in the IdM protocol specification
10. run automated tests provided by pentesting tools
11. inspect test results to verify whether the tested system suffers any vulnerability
12. for each vulnerability reported, use the MSC Drawer to identify where it is exposed
13. once identified, assess the risk linked to the vulnerabilities discovered using STIX Visualizer

5.5 Considerations and Limitations

The implementation of MIG architecture just presented focused on three properties: *modularity*, *flexibility*, and *usability*. Modularity guarantees that each component of MIG is independent from the others. This means that we can remove a component without impacting the functionalities provided by the others and, at the same time, we can add a new component without the need to change the existing environment. Flexibility ensures that MIG components can adapt to the environment where they are executed. For instance, provided tools have to run both on Windows and Unix-like systems and have to provide port customization based on user needs. Finally, usability, hence MIG components have to provide straightforward configuration, automatic deployment and execution, and user-friendly interaction.

Combining what we learned from the literature review process and our experience we decided to adopt the following technologies to achieve these properties: Docker, NodeJS, Burp and Selenium. Docker is a cross-platform application that allows deployment and execution of services on containers. These containers are images that provide virtualization capabilities at the operating system level. They are executed in a sandbox and have no access to the external file system. NodeJS is a JavaScript open-source cross-platform framework that executes JavaScript code outside the browser. In particular, it allows server-side scripting and creation of dynamic web pages therefore it is commonly used to develop web application. Burp is a cross-platform proxy tool providing useful API for inspection and modification of intercepted messages. Finally, Selenium is a browser automation library for automated testing of web applications.

In MIG Backend, all available instances (Table 4.1 and 4.2) are implemented and configured to allow ports customization based on user needs. Each instance runs on Docker that allows deployment execution of services on containers, images that run independently from the operating system. Therefore the instances we provide achieve flexibility. Since the instances belonging to the same protocol are interchangeable without the need of any additional operation, modularity is also achieved. Both properties are ensured also by the other component of MIG Backend: STIX repository. STIX repository is organized as a JSON file therefore it is modular and flexible by design.

The dashboard is a webapp developed in NodeJS. The use of NodeJS allows the execution of the dashboard on every platform and since the port where it runs is customizable, the aforementioned properties are guaranteed. Additionally, the possibility to upload custom IdM system and pentesting tools compliant with the provided guidelines enhances the concept of modularity. The dashboard provides also the usability for MIG Backend and MIG Frontend component. It works as interface for the configuration of the sandbox containing the IdM system and MIG Frontend. Through a form, the user can customize the ports where the services will run, insert custom credentials to authenticate at the IdP, and give a name to the environment he is creating. The form submission starts the automatic generation of the needed files. Once the configuration process is finished, the steps to run the local IdM system in the sandbox and MIG Frontend together with the location of the folder with the files are displayed in the dashboard webapp. The same information is also written in a `README` file added to the environment folder. Therefore the user has only to insert basic information and then the dashboard does everything by itself.

As for MIG Backend and the dashboard, MIG Frontend is also compliant with the three properties we focus on. The proxy we decided to adopt is Burp. This choice has been done because it runs on every operating system and it provides a set of useful and easy-to-use APIs for the development of the pentesting tools. These tools, called extensions in Burp, are `jar` files that can be easily imported. In our case, the provided pentesting tools are automatically loaded when Burp is launched. They are developed in Java and exploit Selenium to automatically execute the tests. Given that, both the proxy and the pentesting tools satisfy the three properties of MIG. They are modular since they can be quickly added and removed without the need of any additional effort. They are flexible since they run on every platform and the proxy port can be customized. And finally, they are easy to use since a simple command launch the proxy and load the pentesting tools, and once the user has inserted the trace, the tests performed are automatically executed.

These properties are satisfied also by MSC Drawer and STIX Visualizer. The two webapps are developed in NodeJS and deployed on Docker containers. In addition, the ports where they expose the services are customizable therefore modularity and flexibility are achieved. Concerning usability, both webapps aim to streamline cumbersome time-consuming processes that usually are performed manually. MSC Drawer simplifies and improves the way of analyzing the HTTP traffic while STIX Visualizer fastens the research of relevant information about vulnerabilities, attacks, and mitigations.

However, there is still something that can be improved. At the moment, the most limited feature is the upload of a custom IdM system. We require the upload of a custom IdM system already configured in Docker because the heterogeneity of the available technologies is difficult to manage without deep knowledge about the adopted solution. Therefore, if on one hand the upload of a IdM system ready for the deployment is more usable since the dashboard does not have to configure anything, on the other hand, we lose a bit on flexibility since we can not integrate and test a custom instance (e.g. a C) against the instances provided by MIG (e.g. one or more of the available IdPs). Nonetheless, the integration process is not as simple as it might seem. Each solution has its own structure (i.e. attributes, endpoints, federation process, etc.) therefore it would be difficult to define a unique pattern to integrate the uploaded entity with the already available ones. But this is not the only issue. Assuming the integration is possible, another problem arises. At the moment of the entity upload, all the configuration needed for the integration has to be also inserted. This means that together with the `zip` file of the uploaded instance, the user has to provide information about attributes, endpoints and all the other aspects that have to be configured at the moment of the deployment. Alternatively, the user might decide to adapt the entity he is uploading to the ones provided by MIG. However, a procedure like this requires different setups when interacting with different technologies.

Therefore, even if the solutions to upload a single entity just described might increase MIG flexibility, they impacted a lot the usability. We think the achievements in flexibility provided by the upload of a single entity does not worth the losses we have in usability. For this reason, at the moment we prefer to ensure usability over flexibility and provide the upload feature for a complete IdM system. Future research and development will focus on the integration of a single entity in a way that does not impact as much as now the usability.

6 Use Cases

MIG can be used in several ways. As previously said, it can be adopted both by professionals and students. In this chapter we present three different use cases. The first describes pentesting activities in the wild on an industrial system implementing a service compliant with the Payment Service Directive 2 (PSD2)¹. The second describes a scenario where we run a system belonging to Italian National Mint and Printing House (IPZS)² in a sandbox and we tested it using MIG. The last use case illustrates how MIG can be used for a hands-on pentesting experience at the university.

6.1 Pentesting an IdM Protocol in the Wild

In the context of a project of Security and Trust (S&T)³ research group of Fondazione Bruno Kessler (FBK)⁴ we have the possibility to test the effectiveness of MIG against a real IdM system in the wild. The service we have to assess belongs to an important Italian identity provider and we have to evaluate the security of its PSD2 implementation. PSD2 is the second Payment Services Directive, designed by the countries of the European Union to regulate payment services throughout the European Union and facilitate secure online payments. Two fundamental aspects of PSD2 are Strong Customer Authentication (SCA) and dynamic linking. The former requires the implementation of an authentication based on two or more factors while the latter ensures that authentication token are linked to a specific payment amount and in case of changes in the payment amount, the linked token becomes invalid. The system we analyzed implements these features using OAuth/OIDC as IdM protocol therefore we used MIG Frontend tools to perform the security assessment.

Following the steps described in Section 5.4, we configure the tools and we start using MSC Drawer. MSC Drawer was configured to intercept and draw the relevant messages regarding OAuth/OIDC traffic. We perform the authentication on the SUT to generate its message sequence chart. The diagram displayed in the MSC Drawer showed that the SUT uses OAuth authorization code flow. An example of a simplified view of the standard authorization code flow is provided in Figure 6.1. As said in Section 2.3 the names of some entities might change from the standard ones (C and IdP) we identify for IdM protocols. In case of OAuth, the IdP is represented by the AS. The following steps describe in details the authorization code flow:

1. **Authorization request:** the C redirects the user agent (usually a browser) to the authorization endpoint at the AS. The C adds to the authorization request its identifier, the scope of the request, the URI where the authorization response will be consumed and an opaque value recommended to keep track of the user session and mitigate CSRF attacks. This information are contained in the parameters `client_id`, `scope`, `redirect_uri` and `state` respectively. The last parameter present in step 1 of Figure 6.1 is `response_type` with the value “code” to indicate that the grant type is authorization code grant.
2. **RO authentication and consent:** the AS authenticates the RO, usually the user, and asks him the consent to let the C access the information needed. Usually at this step the AS shows a consent form containing the information the C wants to access.
3. **Authorization response:** if the RO grants the access, the AS redirects the user back to the C. The message generated is called authorization response and it contains two parameters: `code` and `state`. The former is the authorization code the C will use to obtain the access token at the

¹<https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX:32015L2366>

²<https://www.ipzs.it/>

³<https://stfbk.github.io/>

⁴<https://www.fbk.eu/en/>

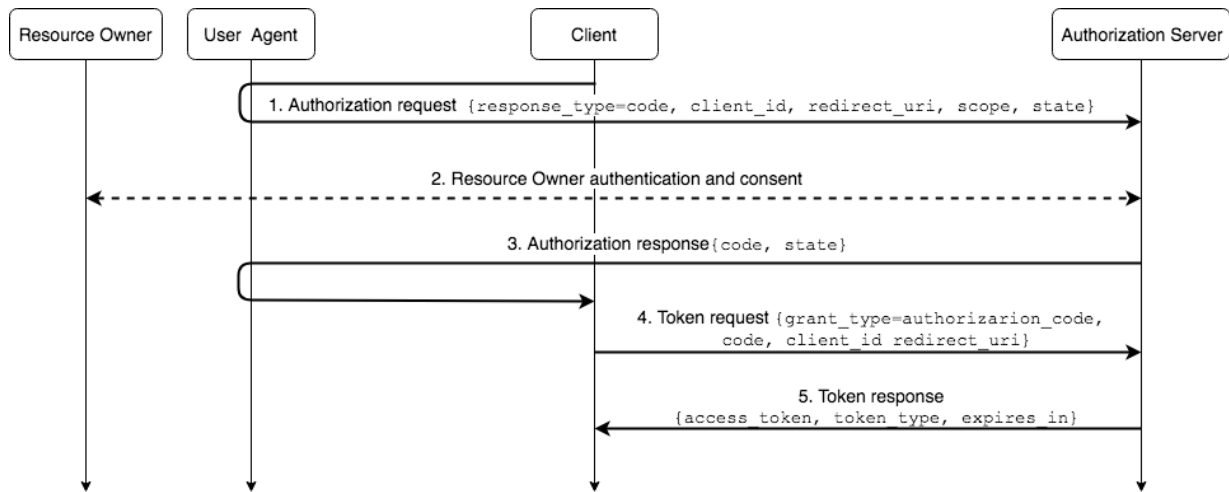


Figure 6.1: OAuth authorization code flow

AS while the latter is the mechanism implemented to maintain the session and should contain the same value of the **state** parameter of the authorization request. The message is sent to the redirect URI specifies in the authorization request.

4. **Token request:** the C requests the access token using the **code** received in the previous step. Together with the code, the C adds to the message its **client_id** and the **redirect_uri** where the received token will be verified.
5. **Token response:** the AS verifies C identity, validates authorization code and redirect URI, and sends back the access token together with the token type and its date of expiration.

The comparison of this flow with the one generated by the intercepted messages immediately highlights an incoherence. There were indeed two messages transported over an insecure HTTP channel. The missing protection of the messages using Transport Layer Security (TLS) [68] exposes the user of the service to several attacks such as man-in-the-middle, eavesdropping and tampering. Therefore, just by the analysis of the message sequence chart displayed in MSC Drawer we were able to spot a relevant flaw in the implementation.

Moving forward, we run the automated tests provided by MIG pentesting tools to verify if there were additional vulnerabilities. The active and passive analysis performed using MIG pentesting tools evidences the following misbehaviours:

- **redirect_uri** parameter: this parameter is not correctly checked during the authentication process on the SUT. **redirect_uri** is set by the C when the request for authentication is generated and sent to the IdP. After the authentication of the user, the IdP redirects the user agent to the URI specified in the value of this parameter. However, if this parameter is not properly checked, as for the SUT, the user might be exposed to several attacks. For instance, an attacker can exploit the missing verification of the **redirect_uri** parameter to make the victim visits a malicious site controlled by him. Doing this, the attacker might gain access to a valid authorization code or access token that he can use to impersonate the victim.
- **state** parameter: this parameter is a random value used to maintain the state between the authorization request and the relative response. It is generated by the C and sent back by the IdP after the authentication of the user. The purpose of this parameter is to prevent CSRF attacks. However, if this protection mechanism is implemented but it is not checked, it becomes useless. This is the case of the SUT where our automated tests discovered that this parameter can be modified and also deleted. Therefore, an attacker that performs a CSRF attack might be able to associate the victim sensitive data to an attacker-controlled account.
- **compliance:** some parameters required by the standard were not present in the implementation of the protocol. There were instead some parameters included in the protocol but not considered

by the involved entities. For instance, the active tests modified and/or removed some of these parameters but the behaviour of the IdP did not change at all. This means these parameters were not properly checked by the IdP.

Thanks to MIG pentesting tools we were able to spot these additional vulnerabilities on the SUT therefore we proved the effectiveness of our solution in a real IdM system in the wild. All the discovered vulnerabilities have been reported to the company and fixed by its developers. URLs have been adjusted to HTTPS and the server has been modified to force the transmission of all HTTP messages over TLS. A verification on the `redirect_uri` value has been implemented and now attackers can not modify its value. Checks on the presence of the `state` parameter and its integrity have been added therefore the mechanism to prevent CSRF that this parameter offers are now correctly implemented. Finally, missing required parameters have been added and unused ones have been removed.

6.2 Pentesting an IdM Protocol in a Sandbox

Thanks to an industrial collaboration between S&T unit and the IPZS we have the opportunity to test the effectiveness of MIG to assist the development of an IdM system. The IdM system in development we had to test uses SAML and implements strong authentication with the support of CIE 3.0, the Italian digital identity card. In the context of authentication using CIE 3.0, identity verification is performed through the X509 certificate contained in the digital identity card. A standard authentication process using SAML is presented in Figure 6.2 and follows these steps:

1. **User asks to access a resource** on the SP. As for OAuth, also in case of SAML there is some changes in the entity names. In particular, the SP is the equivalent of the C we defined in our definition of IdM protocols.
2. **Authentication request**: the SP issues an authentication request to the IdP. This message usually contains two parameters: `SAMLRequest` and `Relaystate`. The former contains the authentication request while the latter is an opaque value that can be used at the discretion of who is implementing the IdM system. For instance it might carry the URL of the resource requested during step 1.
3. **IdP verifies user identity**: the user authenticates at the IdP. Depending on the IdP, different types of authentication can be performed. In the case of IPZS scenario, the user has to provide the X509 certificate contained in his CIE 3.0
4. **Authentication Response**: the IdP issues an authentication response message called assertion. It contains the identity information of the user just authenticated. This data is encoded in the `SAMLResponse` parameter. The `Relaystate` parameter must instead contain the same value of the `Relaystate` parameter at step 2.
5. **SP grants or denies permission to the user**: using the data contained in the received assertion, the SP grants or denies the permission to access the resources requested in step 1 by the user.

Once arrived at the IdP for the authentication, the user has to let the IdP read the certificate of his personal card. The certificate containing the user personal information can be read via the NFC component installed on the card. However, for the sake of our tests, the operation of providing the digital identity card certificate has been simulated via software. Since the original CIE 3.0 can not be used for testing purposes, we generated fake certificates with the same characteristics of the original ones. Concerning IdP and C, we configured them to run on Docker following the guidelines provided by MIG. Then we exploited the upload feature of MIG dashboard to load IPZS IdM system just configured, run it in a sandbox and tested it using MIG Frontend tools. To make the assessment more complete, we upload an additional plugin to test XSW attacks. This tools called SAML Raider has been added to MIG pentesting tools using the dashboard upload feature.

We started the tests using MSC Drawer and we intercepted the messages belonging to SAML standard. These messages were displayed on the MSC Drawer as message sequence chart. The analysis of the

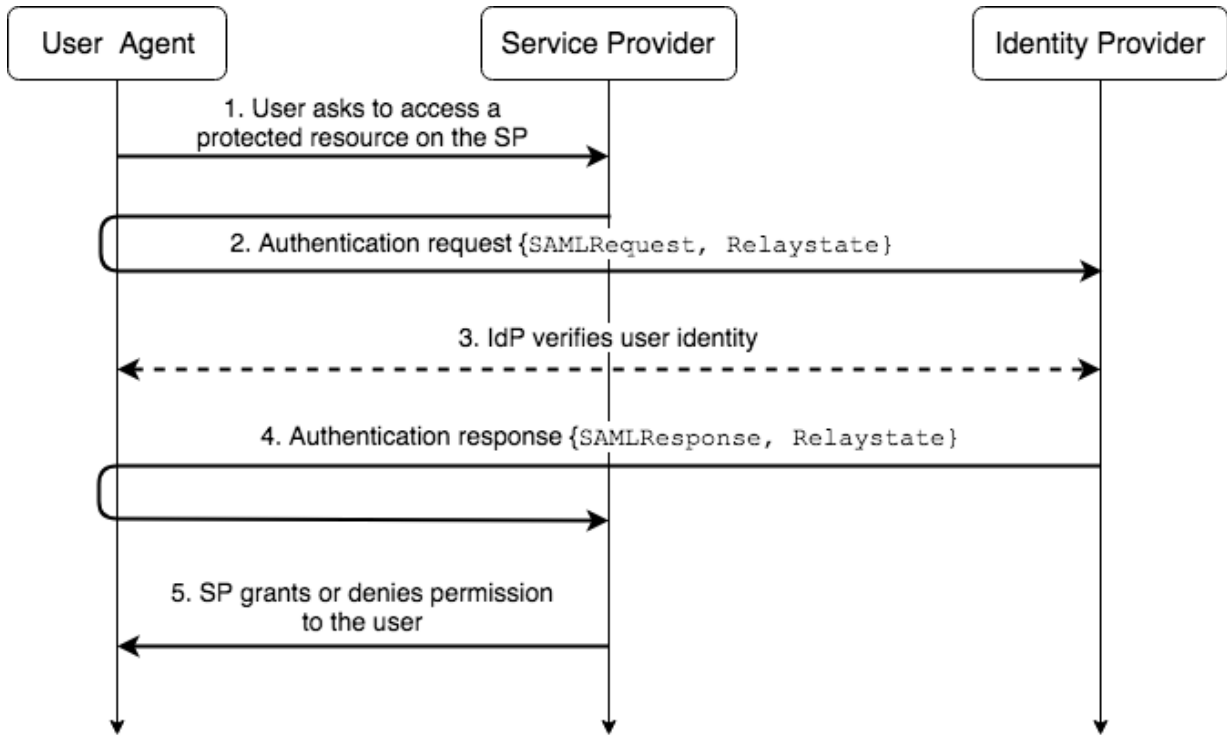


Figure 6.2: SAML authentication flow

intercepted flow confirmed that the implementation of the SUT is compliant with SAML specifications. However, we noticed two additional parameters in the message containing the authentication request sent by the C to the IdP: **Signature** and **SigAlg**. These two parameters are added to the authentication request in the case the original SAML protocol message was signed using an XML digital signature and HTTP Redirect Binding [69] is used for the communication. HTTP Redirect Binding means that SAML message is carried via HTTP GET method. According to SAML Binding specifications, due to URL-size constraints, XML signature is not encoded inside the signed SAML message but it is replaced by **Signature** and **SigAlg** parameters. **SigAlg** is an additional query string parameter and it identifies the algorithm used to sign the SAML protocol message (the authentication request in our case). To construct the signature, a string consisting of the concatenation of the RelayState, SigAlg, and SAMLRequest (or SAMLResponse) is constructed in the following way: **SAMLRequest=value&RelayState=value&SigAlg=value** (in case of SAMLResponse, it replaces SAMLRequest parameter). The resulting string of bytes is the string feed into the signature algorithm. Any other content in the original query string is not included and not signed. Therefore we can assert that a security mechanism ensuring the authentication request can not be tampered is implemented. Nevertheless, we went deeper into the testing process to verify if the signature is properly checked. The modification of the **Signature** parameter value or the **SigAlg** parameter value generates an error. An error is generated also in the case the **Signature** parameter is removed. Instead, in case we remove only the **SigAlg** parameter, the authentication process is not impacted and it proceeds without any error. However, according to the standard, this behaviour makes the tested system not compliant with SAML specification that says “*the signature algorithm identifier MUST be included as an additional query string parameter*” [69]. Therefore, even if the missing presence of **SigAlg** parameter does not lead to any attacks, to be compliant with SAML standard, an additional check on the presence of this parameter should be implemented.

Concerning the automated tests, some misconfigurations have also been spotted. Passive tests evidenced that **OneTimeUse** attribute was not included in the assertion generated by the IdP. The lack of this parameter might lead to a replay attack targeting the C. In this type of attack, a malicious user can consume a valid assertion multiple times [59]. However, to verify if it is actually possible to perform replay attacks, we did some additional verifications. We authenticated on the SUT and we filtered all the traffic through the proxy. When we intercepted the assertion where **OneTimeUse**

attribute is not used, we reused it in a new session. Surprisingly, at the moment when we tried to consume the assertion the second time, the C generates an error saying that an assertion with the same ID has been already consumed. This means that, even if `OneTimeUse` attribute is not used, the C does not allow the consumption of an assertion multiple times. From the point of view of security this is good because the C prevents replay attacks but it also evidences a misconfiguration. Theoretically, the assertion without `OneTimeUse` attribute is supposed to be consumed multiple times but in this case, an assertion can be used only one time due to a check implemented on the C. Even if it is not dangerous, we suggest to fix the misconfiguration with the addition of `OneTimeUse` attribute to the assertion generated by the IdP.

Active tests proved instead that SUT does not implement any checks on the session. This means that the C does not verify if the user agent where the authentication request has been created is the same where the respective response is consumed. Therefore the C can not be sure that the user who has authenticated on the IdP is the same that actually will access to services of the C. An attacker can exploit this implementation flaw to perform Login CSRF similar to the one reported in [14]. Since a user just authenticated on the IdP can consume his assertion on a different session than the actual one, the attacker can craft a message to trick the victim to consume the assertion containing the attacker information. Doing this, the victim will either be authenticated into an attacker controlled account or expose confidential information to the attacker [70, 71, 72]. Even if this vulnerability has been detected during the development phase, considering that IPZS is a company belonging to the Italian public administration, the detected flaw should not be underestimated and a countermeasure to prevent attacks like Login CSRF should be implemented.

To complete the tests, we use SAML Raider to verify if the SUT is vulnerable to any XSW attacks. This plugin has eight test cases for XSW attacks. However, since it is semi-automatic, tests have to be mainly performed manually. For each of the eight attacks, we have to authenticate on the SUT, intercept the assertion generated by the IdP and inject the XSW attacks. In case the assertion is consumed without errors, the C is vulnerable to XSW attacks. Otherwise, it can be considered secure. Fortunately, none of the eight tests has been successful therefore the SUT implements correct mechanisms to prevent XSW attacks.

Summarizing, we used MIG to assist the development of an IdM system based on SAML. Following the MIG guidelines we configure the IdM system we have to test to run on Docker and we then upload it on the dashboard for the set up together with MIG Frontend tools. The assessment we conduct proved the effectiveness of MIG during the development phase. We were able to spot a misconfiguration in the use of `SigAlg` parameter that makes the SUT not compliant with SAML specification. We discover that `OneTimeUse` parameter was not used which might lead to replay attacks. Nevertheless, we also proved that even if it was not used, the C allows the consumption of an assertion only one time. Therefore, we detected a misconfiguration also in the use of this attribute. Finally, we discovered a vulnerability that might expose the user to Login CSRF attacks which could be really severe in case of a production system.

6.3 Hands-on Pentesting Experience at the University

In the last use case we present, MIG is used in a university context for learning and security awareness purposes [73]. After some theoretical classes where IdM protocol are explained to the students, we propose to use MIG tools to put in practice the learned concepts and perform pentesting activities on IdM systems recreated in sandboxes. To exploit all the tools provided by MIG and make students learn as much as possible from the experience, we propose to structure the laboratory session in two parts. The first part to study at high level the SUT using MSC Drawer while the second to go deep in the details using MIG pentesting tools.

Based on standard discussed during the theoretical classes, the professor selects either OAuth/OIDC or SAML, picks a couple of vulnerable C and IdP from the ones available in MIG Backend and uses the dashboard to configure them. After the generation of the necessary files, the professor shares them with the students that, following the provided instructions, can quickly run the sandbox with the IdM system generated and MIG Frontend tools. Once everything is set up and running, the pentesting activities can start. The students login on the SUT to draw the message sequence chart describing the

authentication flow just performed. This operation leads to the first step of the pentesting activities: verifying whether the SUT is compliant with the specifications defined in the standard. To do that, the students have to compare the message sequence chart displayed in the MSC Drawer to the one described in the IdM protocol specification. This quick comparison can be used by the student to spot anomalies on the authentication flow. The second step of this part consists of the use of the features provided by MSC Drawer to check if the parameters required by the standard are present in the messages exchanged between the student browser and the SUT.

The use of a tool like MSC Drawer allows the students to learn the basic notion about the tested IdM protocol. The graphical representation of the exchanged messages in the message sequence chart gives them a concrete understanding of the operations performed to authenticate the user. The inspection of the message content allows them to learn the parameters characterizing each protocol and the values that these parameters usually take.

After this initial phase of analysis with MSC Drawer, students can move forward and start using MIG pentesting tools to perform more fine-grained checks on the SUT. Active and passive tests performed by these tools will give to students a broader view of the vulnerabilities the tested IdM system might suffer. Each vulnerability detected is briefly described in the report available in MIG pentesting tools therefore students have the possibility to learn why the tested configuration is vulnerable. This information can be combined with the CTI information STIX Visualizer provides to get a more detailed understanding of the discovered vulnerability, the attacks that can exploit it and the countermeasure that should be implemented to fix the flaw. Finally, the students can use what they learned from MIG pentesting tools report and STIX Visualizer, and the message sequence chart displayed in the MSC Drawer to identify the entity (either C or IdP) exposing the discovered vulnerability.

Summarizing, the use of MIG in a university lets students learn:

- the authentication flow of the IdM protocol implemented by the SUT in terms of messages exchanged by the user browser, the C and the IdP
- the parameters characterizing the IdM protocol implemented by the SUT
- the most common vulnerabilities IdM protocols might suffer
- the attacks that can be performed on a IdM system suffering a specific vulnerability
- the countermeasures that should be implemented to fix the discovered flaw
- how to identify the entity suffering the vulnerability

7 Conclusion

The proliferation of Internet services requiring password-based authentication to access them their functionalities put the lights on the issue of credential management. Password fatigue due to the large number of user account makes users reuse the same password across multiple platforms or choose simple passwords that can be easily stolen. To tackle this problem, SSO, a new mechanism to authenticate on multiple online services using the same set of credentials, has been introduced. We identified the protocols implementing this authentication schema where a C server relies on a trusted third-party server, the IdP, for user authentication as IdM protocols. Among the existing IdM protocols, the focus of our study has been on SAML and OAuth/OIDC. Even if from the specification point of view these protocols are considered secure, their implementations often lack of some crucial details which make them vulnerable. However, spotting these misconfiguration is not easy and takes lot of times. For this reason, during the years researchers started the development of (semi-)automatic tools to detect vulnerabilities in IdM systems.

As first contribution of this thesis, we analyse the available state of the art of (semi-)automatic pentesting tools. The analysis focus on different types of pentesting tools, from general tools to ones more specific to the tested protocol. Using what we learned from the state of the art, we designed and implemented MIG, the main contribution of this thesis. MIG is a tool to support the creation of sandboxes containing IdM system and perform automated pentesting of IdM protocols. MIG consists of three main components: MIG Backend, MIG Frontend and the dashboard. MIG Backend contains a set of instances of C and IdP used to recreate IdM systems in a sandbox and a repository of CTI information. MIG Frontend provides the tools to support and automate the pentesting activities of IdM protocols. MIG Frontend tools include MSC Drawer to graphically visualize the message sequence chart of the IdM system under test and inspect the message contents, STIX Visualizer to represent in a graph CTI information about vulnerabilities, attacks and mitigations of IdM systems, and two pentesting tools to perform automated tests on the IdM system under test. The tools included in MIG Frontend can be used to test both IdM systems in the wild, which means on the Internet, and in a sandbox in the laboratory. The third component, the dashboard, is used to set up an IdM system in a sandbox and to configure MIG Frontend tools.

The effectiveness of MIG has been verified on three different use cases that are the last contribution of this thesis. The first use case describes the test of an IdM system in the wild belonging to an important Italian identity provider. This IdM system uses OAuth/OIDC to implement the strong authentication required by PSD2 standard. However, the tests we conduct evidences the presence of some misconfiguration in the implementation. Two parameters (`redirect_uri` and `state`) were not correctly checked therefore the system was exposed to potentially severe attacks. Additionally we discovered that some messages were not exchanged over HTTPS and there were some parameters not supposed to be present. These findings proved that MIG is useful to test a real IdM system in the wild.

In the second use case we present, we used MIG to recreate in a sandbox an industrial IdM system in development belonging to IPZS and to test it. In this case, the IdM system uses SAML. Following the guidelines of MIG, we configure it to run in a sandbox and we tested it using MIG Frontend tools. The results highlight the presence of two misconfiguration and a severe vulnerability in the tested implementation thus proving the effectiveness of our solution to support developers during the implementation of IdM protocols.

Finally, the third use cases concerns the use of MIG for hands-on pentesting experience at the university. We propose to recreate an IdM system in a sandbox using the provided instances to let students perform pentesting activities and learn IdM protocols, their vulnerabilities, the attacks that can be performed and the mitigations to prevent them.

Starting from what we did in this thesis, several future work can be done to improve the actual solution. The first one concerns the expansion of the available instances. Even if the technologies we provide represent part of the most famous implementation recognized nowadays, in the future this list could be improved. However, to do that more research has to be done. A way to integrate the new solutions with the ones already available without losing the usability we offer has to be found. As previously said, this is not simple due to the heterogeneity of the technologies. Future researches have to focus on the definition of a common pattern to integrate new solutions with the existing ones. Another future work concerns MIG pentesting tools. Actually they offer a quite complete set of tests but they can still be improved. The possibility to expand the actual set of tests and integrate new ones is fundamental to keep the pace of new vulnerabilities proliferation. Moreover, the possibility to integrate our pentesting tools with others available on the Internet would contribute to increase the effectiveness of MIG, both in the case it is used in the wild and in the laboratory. Finally increasing the available CTI information will help developers and tester to have a better understanding of vulnerabilities of IdM system, attacks that can exploit them and possible mitigation to decrease the risk, and students to increase their security awareness on the different aspects concerning IdM protocols.

Bibliography

- [1] The Past, Present, and Future of Password Security. <https://www.mcafee.com/blogs/consumer/consumer-threat-notice/security-world-password-day/>. Last access on November 23, 2020.
- [2] Password Reuse Abounds, New Survey Shows. <https://www.darkreading.com/informationweek-home/password-reuse-abounds-new-survey-shows/d/d-id/1331689>. Last access on November 23, 2020.
- [3] Security Assertion Markup Language (SAML) V2.0 Technical Overview. <https://docs.oasis-open.org/security/saml/Post2.0/sstc-saml-tech-overview-2.0.html>. Last access on November 23, 2020.
- [4] The OAuth 2.0 Authorization Framework. <https://tools.ietf.org/html/rfc6749>. Last access on November 23, 2020.
- [5] OpenID Connect. <https://openid.net/connect/>. Last access on November 23, 2020.
- [6] Armando, Alessandro and Carbone, Roberto and Compagna, Luca and Cuellar, Jorge and Tobarra, Llanos. 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 '08, pages 1–10, New York, NY, USA, 2008. Association for Computing Machinery.
- [7] Armando, Alessandro and Carbone, Roberto and Compagna, Luca and Cuéllar, Jorge and Pellegrino, Giancarlo and Sorniotti, Alessandro. An Authentication Flaw in Browser-Based Single Sign-On Protocols: Impact and Remediations. *Comput. Secur.*, 33:41–58, March 2013.
- [8] Fett, Daniel and Küsters, Ralf and Schmitz, Guido. The Web SSO Standard OpenID Connect: In-depth Formal Security Analysis and Security Guidelines. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pages 189–202, 2017.
- [9] Pai, Suhas and Sharma, Yash and Kumar, Sunil and Pai, Radhika M. and Singh, Sanjay. Formal Verification of OAuth 2.0 Using Alloy Framework. In *Proceedings of the 2011 International Conference on Communication Systems and Network Technologies*, CSNT '11, pages 655–659, USA, 2011. IEEE Computer Society.
- [10] Sun, San-Tsai and Beznosov, Konstantin. 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*, pages 378–390, 2012.
- [11] Zhou, Yuchen and Evans, David. SSOScan: automated testing of web applications for single sign-on vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 495–510, 2014.
- [12] Sumongkayothin, Karin and Rachtrachoo, Pakpoom and Yupuech, Arnuphap and Siriporn, Kasidit. OVERSCAN: OAuth 2.0 Scanner for Missing Parameters. In *International Conference on Network and System Security*, pages 221–233. Springer, 2019.

- [13] Li, Wanpeng and Mitchell, Chris J and Chen, Thomas. OAuthGuard: Protecting User Security and Privacy with OAuth 2.0 and OpenID Connect. pages 35–44, 2019.
- [14] Sudhodanan, Avinash and Carbone, Roberto and Compagna, Luca and Dolgin, Nicolas and Armando, Alessandro and Morelli, Umberto. Large-scale analysis & detection of authentication cross-site request forgeries. In *2017 IEEE European symposium on security and privacy (EuroS&P)*, pages 350–365. IEEE, 2017.
- [15] Password Dos and Dents. <https://krebsonsecurity.com/password-dos-and-dents/>. Last access on November 23, 2020.
- [16] The Password Expose. <https://lp-cdn.lastpass.com/lporcamedia/document-library/lastpass/pdf/en/LastPass-Enterprise-The-Password-Expose-Ebook-v2.pdf>. Last access on November 23, 2020.
- [17] Online Security Survey. https://services.google.com/fh/files/blogs/google_security_infographic.pdf. Last access on November 23, 2020.
- [18] Armando, Alessandro and Carbone, Roberto and Compagna, Luca. LTL Model Checking for Security Protocols. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*, pages 385–396, 2007.
- [19] Chari, Suresh and Jutla, Charanjit and Roy, Arnab. Universally Composable Security Analysis of OAuth v2.0. *IACR Cryptology ePrint Archive*, 2011:526, 01 2011.
- [20] Li, Xiaowei and Xue, Yuan. BLOCK: a black-box approach for detection of state violation attacks towards web applications. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 247–256, 2011.
- [21] Xing, Luyi and Chen, Yangyi and Wang, XiaoFeng and Chen, Shuo. InteGuard: Toward Automatic Protection of Third-Party Web Service Integrations. In *NDSS*, 2013.
- [22] Wang, Rui and Chen, Shuo and Wang, XiaoFeng. Signing me onto your accounts through facebook and google: A traffic-guided security study of commercially deployed single-sign-on web services. In *2012 IEEE Symposium on Security and Privacy*, pages 365–379. IEEE, 2012.
- [23] Bai, Guangdong and Lei, Jike and Meng, Guozhu and Venkatraman, Sai Sathyanarayan and Saxena, Prateek and Sun, Jun and Liu, Yang and Dong, Jin Song. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. 2013.
- [24] Shernan, Ethan and Carter, Henry and Tian, Dave and Traynor, Patrick and Butler, Kevin. More guidelines than rules: CSRF vulnerabilities from noncompliant OAuth 2.0 implementations. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 239–260. Springer, 2015.
- [25] Yang, Ronghai and Li, Guanchen and Lau, Wing Cheong and Zhang, Kehuan and Hu, Pili. Model-based security testing: An empirical study on oauth 2.0 implementations. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 651–662, 2016.
- [26] Philippaerts, Pieter. OAuch: Analyzing the Security Best Practices in the OAuth 2.0 Ecosystem. <https://www.youtube.com/watch?v=eaFQFm1K5yI/>. Last access on November 23, 2020.
- [27] OSW 2020. <https://osw2020.com/>. Last access on November 23, 2020.
- [28] Mainka, Christian and Mladenov, Vladislav and Schwenk, Jörg. Do not trust me: Using malicious IdPs for analyzing and attacking Single Sign-On. In *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 321–336. IEEE, 2016.

- [29] Christopher Späth, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. SoK: XML Parser Vulnerabilities. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association.
- [30] OpenID FAPI conformance suite. <https://gitlab.com/openid/conformance-suite>. Last access on November 23, 2020.
- [31] Financial-grade API (FAPI). <https://fapi.openid.net/>. Last access on November 23, 2020.
- [32] Mainka, Christian and Somorovsky, Juraj and Schwenk, Jörg. Penetration testing tool for web services security. In *2012 IEEE Eighth World Congress on Services*, pages 163–170. IEEE, 2012.
- [33] SAML Raider. <https://github.com/CompassSecurity/SAMLRaider>. Last access on November 23, 2020.
- [34] Mainka, Christian and Mladenov, Vladislav and Guenther, Tim and Schwenk, Jörg. Automatic Recognition, Processing and Attacking of Single Sign-On Protocols with Burp Suite. Gesellschaft für Informatik eV, 2015.
- [35] Juraj Somorovsky, Andreas Mayer, Jörg Schwenk, Marco Kampmann, and Meiko Jensen. On breaking saml: Be whoever you want to be. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 397–412, 2012.
- [36] Billion laughs attack. <https://en.wikipedia.org/wiki/BillionLaughsAttack>. Last access on November 23, 2020.
- [37] Jon Barber. SAMLyze. <https://www.blackhat.com/us-15/arsenal.html#jon-barber>. Last access on November 23, 2020.
- [38] Calzavara, Stefano and Focardi, Riccardo and Maffei, Matteo and Schneidewind, Clara and Squarcina, Marco and Tempesta, Mauro. WPSE: Fortifying Web Protocols via Browser-Side Security Monitoring. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1493–1510, 2018.
- [39] Google Vulnerability Reward Program (VRP). <https://www.google.com/about/appsecurity/reward-program/>. Last access on November 23, 2020.
- [40] Facebook Whitehat. <https://www.facebook.com/whitehat>. Last access on November 23, 2020.
- [41] Microsoft Bug Bounty Program. <https://www.microsoft.com/en-us/msrc/bounty>. Last access on November 23, 2020.
- [42] Dashevskiy, Stanislav and Dos Santos, Daniel Ricardo and Massacci, Fabio and Sabetta, Antonino. TESTREX: a Testbed for Repeatable Exploits. In *7th Workshop on Cyber Security Experimentation and Test (CSET 14)*, 2014.
- [43] Nilson, Gary and Wills, Kent and Stuckman, Jeffrey and Purtilo, James. BugBox: A Vulnerability Corpus for PHP Web Applications. In *6th Workshop on Cyber Security Experimentation and Test (CSET 13)*, 2013.
- [44] WebGoat. <https://owasp.org/www-project-webgoat/>. Last access on November 23, 2020.
- [45] OAuth 2.0 Playground. <https://www.oauth.com/playground/>. Last access on November 23, 2020.
- [46] Google OAuth 2.0 Playground. <https://developers.google.com/oauthplayground/>. Last access on November 23, 2020.
- [47] Andrea Bisegna, Roberto Carbone, Ivan Martini, Valentina Odorizzi, Giulio Pellizzari, and Silvio Ranise. Micro-Id-Gym: Identity Management Workouts with Container-Based Microservices. In *International Journal of Information Security and Cybercrime (IJISP)*, Volume 8, Issue 1.

- [48] SAML Open Source Implementations. <http://saml.xml.org/wiki/saml-open-source-implementations>. Last access on November 23, 2020.
- [49] Certified OpenID Connect Implementations. <https://openid.net/developers/certified/>. Last access on November 23, 2020.
- [50] OAuth Services. <https://oauth.net/code/>. Last access on November 23, 2020.
- [51] Duo Finds SAML Vulnerabilities Affecting Multiple Implementations. <https://duo.com/blog/duo-finds-saml-vulnerabilities-affecting-multiple-implementations>. Last access on November 23, 2020.
- [52] SAML Open Source Implementations. <https://spring.io/projects/spring-security-saml>. Last access on November 23, 2020.
- [53] Keycloak. <https://github.com/keycloak>. Last access on November 23, 2020.
- [54] MITREid Connect. <https://github.com/mitreid-connect/>. Last access on November 23, 2020.
- [55] Open redirect in rfc6749 aka 'The OAuth 2.0 Authorization Framework'. <https://blog.intothesyymetry.com/2015/04/open-redirect-in-rfc6749-aka-oauth-20.html>. Last access on November 23, 2020.
- [56] OAuth 2.0 Security Best Current Practice. <https://tools.ietf.org/id/draft-ietf-oauth-security-topics-08.html>. Last access on November 23, 2020.
- [57] Shibboleth IdP. <https://wiki.shibboleth.net/confluence/display/IDP30/Home>. Last access on November 23, 2020.
- [58] Lorenzo Tait. A customized threat modeling for secure deployment and pentesting of saml sso solutions. Bachelor's thesis, University of Trento, 2019.
- [59] Security and Privacy Considerations for the OASIS Security Assertion Markup Language (SAML) V2.0. <https://docs.oasis-open.org/security/saml/v2.0/saml-sec-consider-2.0-os.pdf>. Last access on November 23, 2020.
- [60] Proof Key for Code Exchange by OAuth Public Clients. <https://tools.ietf.org/html/rfc7636>. Last access on November 23, 2020.
- [61] Davide Piva. Assisting developers in securing oauth 2.0 deployment. Bachelor's thesis, University of Trento, 2019.
- [62] OWASP Secure Headers Project. <https://owasp.org/www-project-secure-headers/>. Last access on November 23, 2020.
- [63] Insecure Transport. https://owasp.org/www-community/vulnerabilities/Insecure_Transport. Last access on November 23, 2020.
- [64] Daniel Fett, Pedram Hosseyni, and Ralf Küsters. An extensive formal security analysis of the openid financial-grade api. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 453–471. IEEE, 2019.
- [65] SPID. <https://www.spid.gov.it/>. Last access on November 23, 2020.
- [66] XML External Entity (XXE) Processing. [https://owasp.org/www-community/vulnerabilities/XML_External_Entity_\(XXE\)_Processing](https://owasp.org/www-community/vulnerabilities/XML_External_Entity_(XXE)_Processing). Last access on November 23, 2020.
- [67] STIX Version 2.1 - Committee Specification. <https://docs.oasis-open.org/cti/stix/v2.1/cs01/stix-v2.1-cs01.html>. Last access on November 23, 2020.

- [68] The Transport Layer Security (TLS) Protocol Version 1.2. <https://tools.ietf.org/html/rfc5246>. Last access on November 23, 2020.
- [69] Bindings for the OASIS Security Assertion Markup Language (SAML) V2.0. <https://docs.oasis-open.org/security/saml/v2.0/saml-bindings-2.0-os.pdf>. Last access on November 23, 2020.
- [70] Barth, Adam and Jackson, Collin and Mitchell, John C. Robust Defenses for Cross-Site Request Forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 75–88, New York, NY, USA, 2008. Association for Computing Machinery.
- [71] Lundeen, Rich. The Deputies are Still Confused. <https://media.blackhat.com/eu-13/briefings/Lundeen/bh-eu-13-deputies-still-confused-lundeen-wp.pdf>. Last access on November 23, 2020.
- [72] Grossman, Jeremiah. I used to know what you watched, on YouTube. <https://blog.jeremiahgrossman.com/2008/09/i-used-to-know-what-you-watched-on.html>. Last access on November 23, 2020.
- [73] Andrea Bisegna, Roberto Carbone, Giulio Pellizzari, and Silvio Ranise. Micro-Id-Gym: a Flexible Tool for Pentesting Identity Management Protocols in the Wild and in the Laboratory. In *International Workshop on Emerging Technologies for Authorization and Authentication*. Springer, 2020.