Guardians of the Clouds: When Identity Providers Fail

Andreas Mayer
Adolf Würth GmbH & Co. KG
Künzelsau-Gaisbach, Germany
andreas.mayer@wuerth.com

Marcus Niemietz, Vladislav Mladenov[*], Jörg Schwenk
Horst Görtz Institute for IT-Security
Bochum, Germany
firstname.lastname@rub.de

ABSTRACT

Many cloud-based services offer interfaces to Single Sign-On (SSO) systems. This helps companies and Internet users to keep control over their data: By using an Identity Provider (IdP), they are able to enforce various access control strategies (e.g., RBAC) on data processed in the cloud.

On the other hand, IdPs provide a valuable single point of attack: If the IdP can be compromised, *all* cloud services are affected, including well-protected applications such as Google Apps and Salesforce. This increases the impact of the attack by several orders of magnitude.

In this paper, we analyze the security of six real-world SAML-based IdPs (OneLogin, Okta, WSO2 Stratos, Cloudseal, SSOCircle, and Bitium) which are used to protect cloud services. We present a novel attack technique (ACS Spoofing), which allows the adversary to successfully impersonate the victim in four of these SSO systems. To complete our survey on IdP security, we additionally evaluated the security of these six IdPs against well-known web attacks, and we were successful against four of them. In summary, we were able to break all six SSO systems.

We present a online penetration test tool, *ACSScanner*[1], which is able to detect ACS Spoofing vulnerabilities on arbitrary IdPs. Additionally, we discuss several countermeasures for each attack type, ranging from simple whitelisting to the signing of authentication requests, and from anti-CSRF tokens and HTTP-Only cookies to cookie-TLS-bindings. We have implemented a combination of two advanced countermeasures.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Unauthorized access

[*]The author was supported by the SkIDentity project of the German Federal Ministry of Economics and Technology (BMWi, FKZ: 01MD11030).

[1]`http://ssoattacks.org:8080/acsscanner/`

CCSW'14, November 7, 2014, Scottsdale, Arizona, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3239-2/14/11 ...$15.00.
http://dx.doi.org/10.1145/2664168.2664171.

General Terms

Security

Keywords

Holder-of-Key; identity theft; SAML; SSO; web security

1. INTRODUCTION

Access to cloud services can either be managed directly through a web interface, or by using one of the APIs offered by the cloud service. In the first case, users are typically identified though the combination of username/password. This aspect is not only cumbersome but also seriously insecure, as users frequently choose weak (easy to remember) passwords and/or reuse them on different websites [17]. Moreover, password-based solutions are susceptible to phishing and web spoofing attacks.

Using an API for authentication and authorization gives more flexibility (e.g., Role Based Access Control) and offers the possibility to get better security. The Security Assertion Markup Language (SAML) [8] is the industry standard for such an API, and is offered by many cloud services (e.g., Google Apps, Salesforce, Cisco WebEx, and Workday). SAML defines the exchange of authentication and authorization statements in security tokens called *assertions*. This standardized API is used to facilitate browser-based Single Sign-On (SSO) to tackle the usability, management, and security issues of classical password-based login.

With browser-based SSO (cf. Figure 1) the user (U) authenticates once to a trusted party called the Identity Provider (IdP). Subsequently, U gains access to all federated cloud services, referred to as Service Providers (SPs), without being prompted with another login dialog. The user participates in the SSO flow with his user agent (UA), which is usually a web browser.

The security of an SSO system is only as strong as the security of its weakest part. SSO systems combine HTTP, HTML, JavaScript, and XML technologies. Thus, SSO offers an attractive target to attackers: A security bug in the IdP implementation based on one of these technologies may allow to access all federated cloud services and websites.

ATTACKS. We present successful attacks on six prominent cloud-based IdPs (OneLogin [34], Okta [33], SSOCircle [43], WSO2 Stratos [50], Cloudseal [12], and Bitium [6]), using two different attack classes:

1. *Class 1: AssertionConsumerService (ACS) Spoofing (Section 6).* We describe a novel attack against the

Table 1: Results of our practical evaluation.

SSO System	Website	Affected SPs	ACS Spoofing	Cookie theft	Common Vulnerabilities and Exposures (CVE)
OneLogin	`www.onelogin.com`	≈3,600	✓	✓	CVE-2012-4962, -4963
Okta	`www.okta.com`	≈3,000	–	✓	CVE-2013-0114
WSO2 Stratos	`www.wso2.org`	Open[a]	✓	–	CVE-2012-4961
Cloudseal	`www.cloudseal.com`	Open[b]	–	✓	Assignment in process
SSOCircle	`www.ssocircle.com`	Open[b]	✓	✓	CVE-2013-0115, -0116, -0117
Bitium	`www.bitium.com`	≈1,750	✓	–	Direct communication

[a] All SPs that accept SAML assertions issued by these IdPs are affected.
[b] All federated SPs are affected.

SAML Web SSO Profile [9]. This attack allows the adversary to redirect the security token issued by the IdP to himself, and thus to impersonate the victim to every federated SP. The only prerequisite for this attack is that the victim has to visit a webpage controlled by the adversary. We show the practical feasibility of this attack in four popular SAML-based SSO solutions (cf. Table 1).

2. *Class 2: Attacks against the IdP web application (Section 7).* We discovered multiple cross-site scripting (XSS) [54] vulnerabilities in all six real-world IdPs. These flaws allowed us (in two cases only in combination with a UI redressing attack) to steal the victim's HTTP session cookies set by four IdPs (cf. Table 1), resulting in a complete identity theft of the victim. Again, the victim only has to visit a webpage controlled by the adversary, and in two cases (Okta and Cloudseal) to perform a few additional mouse clicks or drag-and-drop actions on this webpage.

The attacks presented break SSO systems even if strong multi-factor authentication (MFA) is deployed: MFA only protects the initial authentication with the IdP, but we target the system only after the user is authenticated. If a user is not yet logged in to the IdP, we may simply masquerade the attacker's web servers as a Service Provider: Since the normal workflow in this case is to authenticate to the IdP first (possibly using MFA), no doubts will be raised by the victim.

Results, Impact, and Responsible Disclosure. In summary, all six evaluated IdP systems exposed severe security flaws (cf. Table 1). All IdPs are high-profile IdPs with thousands of customers (e.g., London Gatwick Airport, Western Union, Groupon, LinkedIn). They are be used by millions of users to authenticate to many security critical cloud services (e.g., Google Apps and Salesforce).

We promptly reported all vulnerabilities found to the liable security teams as well as to the leading Computer Emergency Response Team (CERT) at Carnegie Mellon University [10]. All parties have acknowledged and fixed the reported vulnerabilities.

Countermeasures. Unfortunately, all known countermeasures to mitigate ACS Spoofing (including those proposed by the SAML standard) have a high administrative overhead, are incompatible with each other, or may be subject to advanced attacks (Section 8.1). To protect against XSS, several best practice technologies exist, but are rarely deployed. None of these technologies offer full protection, since workarounds are known in each case (Section 8.2).

To mitigate *both* attack classes simultaneously, we investigate the standardized, but rarely deployed, OASIS Holder-of-Key (HoK) Profile [28]. HoK binds each SAML assertion to the TLS [14] client certificate of the browser. We extend this strong cryptographic binding to a more holistic approach and additionally bind the HTTP session cookies of IdP and SP to the same client certificate (along the lines of [15]). Please note that client certificates may be self-signed, thus we do *not* rely on a PKI. Our extended approach can, in contrast to [15], be applied without changing existing infrastructures (e.g., web server and web browser) and with minimal performance impact (see Section 8.3).

Contribution. Our main achievements can be summarized as follows:

- *Novelty.* We describe a novel and high-impact attack on SAML-based SSO services (ACS Spoofing), resulting from a logical flaw in the information flow. Furthermore, to our best knowledge, we are the first that analyze the security of cloud-based IdPs against XSS/UI redressing attacks and evaluate the impact of such attacks within the SSO based authentication.

- *Impact.* We show that IdPs that are not vulnerable to ACS Spoofing may still be broken with a combination of XSS and UI redressing attacks. Thus by compromising all six investigated IdPs we show that IdP security must be improved, since an IdP vulnerability affects all federated SPs.

- *ACS Scanner.* Based on our crucial findings, we developed a ACS Spoofing penetration test tool, which is online available. ACS Scanner can be configured via the web interface and it automatically tests the security of the IdP. Therefore, customers and developers can easily test if their IdP is vulnerable against ACS Spoofing.

- *Mitigation.* We investigate several countermeasures against ACS Spoofing for their practical applicability. We broaden the scope of the HoK approach to include HTTP session cookies along the lines of [15], and present a proof-of-concept implementation of this concept for the popular open source framework SimpleSAMLphp [47], together with a performance evaluation.

Outline. The rest of the paper is organized as follows. Related work is given in Section 2. In Section 3 we will briefly introduce SSO, SAML, and HTTP session cookies.

The adversarial model of our attacks is explained in Section 4. Next, we introduce six prominent SSO systems targeted by our analyzes and explain our attack methodology in Section 5. The ACS Spoofing attack, ACS Scanner, and a practical evaluation of vulnerable real-world IdPs are presented in Section 6. Results of combined XSS/UI redressing attacks are supplied in Section 7. Several countermeasures against ACS Spoofing and cookie theft along with our preferred countermeasure are discussed in Section 8. Finally, we conclude and propose future research directions.

2. RELATED WORK

Single-Sign-On (SSO). A variety of vulnerabilities have been identified in browser-based SSO protocols that emerged over the last two decades.

Research on SSO security started with Rubin [29] and Slemko [41] analyzing Microsoft Passport, and Groß [20] describing several attacks for the SAML Browser Artifact Profile. Later, Groß analyzed a revised version of SAML, finding a need for improvement once again [21]. Similar flaws have also been found in the Liberty Single Sign-On protocol [39].

In 2008, Armando et al. [3] built a formal model of the SAML V2.0 Web SSO Profile [9] and have analyzed it with the model checker SATMC. By introducing a malicious SP, they have found a practical attack on the SAML implementation of Google Apps. Same authors have identified another attack on the SAML-based SSO of Google Apps in 2011 [2]. The attack model is related to our ACS Spoofing attack. However, we exploit a logical flaw in the IdP's SAML interface implementation instead of SP-based XSS flaws.

Somorovsky et al. [42] have published an in-depth analysis of XML Signature usage in 14 major SAML frameworks and showed that 11 of them, including Salesforce, Shibboleth, and IBM XS40, had critical XML Signature wrapping flaws. Please note that their results do not have any overlap with our results. They targeted the XML data structure of SAML assertion, whereas we are looking at information flow and web security issues.

Wang et al. [48] analyzed the security of commercially deployed SSO solutions. The authors identified eight serious logic flaws in high-profile IdPs and SP websites (e.g., GoogleID, Facebook, and JanRain), which have allowed an adversary to sign-in as the victim user. They concentrated on the message flow in REST-based authentication protocols like OpenID and did not consider SAML and web application attacks like XSS and UI redressing. Later on, the same authors developed a tool named *InteGuard* detecting invariance in the communication and preventing logical flaws in SPs [51]. InteGuard is not able to mitigate the attacks presented in this paper, since all HTTP messages between the adversary and the SP are valid and do not show abnormalities.

In 2012 Sun and Beznosov [46] screened 96 OAuth SPs for known attacks. They discovered several vulnerabilities caused by implementation flaws.

Recently, Bai et al. [22] have proposed *AuthScan*, a framework to automatically extract the authentication protocol specifications from implementations. They have found multiple security flaws in existing SPs using several important SSO protocols (e.g., Facebook Connect, BrowserID, and Windows Live Messenger Connect). However, they have not investigated SAML-based SSO systems. In 2014 Evans et al. [52] published a tool *SSOScan* evaluating fully automated the security of OAuth 2.0 implementations. However, *SSOScan* does not support the security analysis of SAML based SSO systems and does not cover web attacks like XSS, CSRF, and Clickjacking.

TLS Channel Bindings. In 2007 Karlof et al. [27] have proposed to strengthen the web browser's Same Origin Policy (SOP) by taking the TLS server certificate into account. The proposed *strong-locked SOPs* (SLSOP) tags session cookies with the server's public key and solely returns them to servers if the public key of the server's TLS certificate matches. A similar approach called *Web Server Key Enabled Cookies* (WSKECookies) has been presented by Masone et al. [31].

The use of client certificates in SSO systems was first discussed in [19] and standardized in [28]. Dietz et al. [15] have proposed origin-bound certificates (OBC), an approach aimed at strengthening client authentication for the Web with the use of a TLS extension. OBC require fundamental changes of the TLS stack in web server and browser. The authors are (up to our knowledge) the first to discuss HTTP cookie bindings to TLS client certificates in detail.

Cross-Site Scripting and UI-Redressing. According to OWASP Top 10 2013 [36], cross-site scripting (XSS) is the most prevalent security flaw in today's web applications. XSS vulnerabilities are based on improperly filtered data (e.g., script code) that is injected in a webpage and sent to the browser. Afterwards, the injected malicious content is executed in the victim's browser. XSS has been discovered in 2000 [11] and is categorized into five classes: reflected, persistent, DOM-based, self, and mutation-based XSS [26]. Recently, Heiderich et al. have demonstrated that an adversary can inject *scriptless* data to execute XSS attacks [25].

In 2002 Ruderman [40] pointed out that it is possible to make webpage elements transparent and that a victim can click on these elements without getting any notice. This attack was called UI redressing. In 2008 Hansen and Grossmann have presented clickjacking [23], which is based on UI redressing. They have shown that an adversary could use transparent iFrames in combination with the Adobe Flash Player to get access to the camera and microphone of the victim by hijacking a few clicks. UI redressing attacks can be combined with XSS to make them more powerful (cf. Section 7).

3. FOUNDATIONS

3.1 Single Sign-On

A high-level description of Single Sign-On (SSO) is given in Figure 1. User U directs his web browser (UA) to visit Service Provider SP (1). SP issues a token request (2) to UA (3), who forwards it to the Identity Provider IdP (4). After verifying the clients identity, IdP issues a token for U containing several claims (e.g., user's identity, expiration time, and access rights). In order to guarantee authenticity and integrity of these claims, the token is signed (5). Subsequently, the token is sent to UA (6), who forwards it to SP (7). SP validates the signature and, in case of success, grants access to the protected service or resource of SP. This access-control decision relies on the claims in the validated token. Please note that the whole conversation typically is secured only through server-authenticated TLS channels.

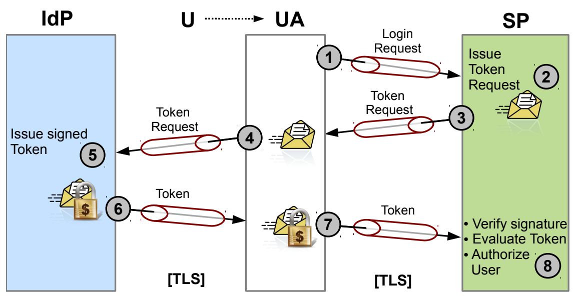


Figure 1: A typical browser-based SSO scenario: The user visits the SP (1), which generates a request token (2+3). He redirects this token to the IdP (4). The issued identity token (5) is sent to the user (6) and forwarded to the SP (7). The whole conversation is secured through TLS channels.

a)

```
<AuthnRequest ID Version IssueInstant
 AssertionConsumerServiceURL?>
  <Issuer>?
  <Subject>?
  <NameIDPolicy>?
  <Extensions>?
  <Signature>?
  <Conditions>?
  <RequestedAuthnContext>?
  <Scoping>?
</AuthnRequest>
```

b)

```
<Assertion Version ID IssueInstant>
  <Issuer>
  <Signature>?
  <Subject>?
    <SubjectConfirmation Method>*
      <SubjectConfirmationData
        InResponseTo? Recipient?>?
  <Conditions>?
  <Advice>?
  <AuthnStatement>*
  <AuthzDecisionStatement>*
  <AttributeStatement>*
</Assertion>
```

Figure 2: `<AuthnRequest>` and `<Assertion>` structure (regular expression notation).

3.2 SAML

The Security Assertion Markup Language (SAML) is a widely-used XML-based standard for exchanging authentication and authorization statements about *subjects*. These statements are contained in security tokens called *assertions*. SAML consists of three other building blocks: (1) *protocols* define how assertions are exchanged between the actors; (2) *bindings* specify how to embed assertions into transport protocols (e.g., HTTP or SOAP) and (3) *profiles* define the interplay of assertions, protocols, and bindings that are necessary for the needs of a specific use case to be met. The investigated SAML Web Browser SSO Profile [9] is the most commonly deployed application scenario.

In order to request an assertion, SAML defines the `<AuthnRequest>` message (i.e. token request). The simplified structure is shown in Figure 2a).

Mandatory attributes are: `Version` states the SAML version in use, `ID` supplies the unique and randomly chosen request identifier, and the time of issuing is identified in `IssueInstant`. The optional `AssertionConsumerServiceURL` (ACS_{URL}) attribute specifies the endpoint URL to which the IdP *must* deliver the issued assertion. The `<Issuer>` element includes the EntityID of the SP. The `<Subject>`, `<NameIDPolicy>` and `<Extensions>` elements are seldom used and are irrelevant for the paper. The authentication request may be protected by a digital signature (`<Signature>`) following the XML Signature standard. In practice, signing is used rarely.

The structure of an assertion (i.e. token), which is issued for the requesting SP, is defined in Figure 2b). The `Version`, `ID`, and `IssueInstant` attributes are required, having the same purpose as in the `<AuthnRequest>`. The `<Issuer>` element specifies the SAML authority (the IdP) that certifies the claim(s). `<Subject>` defines the principal about whom all statements within the assertion are made. `<SubjectConfirmation>` is an optional element utilized for specifying methods (e.g., bearer or HoK) that the SP has to use to verify the validity of the assertion. `<SubjectConfirmationData>` specifies constraints or data required for the subject confirmation. The value of `InResponseTo` *must* match the `ID` of the `<AuthnRequest>`, if a correlation of the assertion to the request is to be made. To prevent malicious forwarding of assertions to unintended recipients, the optional `Recipient` attribute and the ACS_{URL} of the request *must* be equal. The `<*Statement>` and `<Advice>` elements are optional and not relevant for this paper. To assure the integrity and authenticity of the claims made, the whole `<Assertion>` *must* be protected by an XML Signature.

3.3 Session Management with Cookies

HTTP is a stateless protocol. Regarding the authentication a user would be forced to re-enter his login information repeatedly, since the web server treats every HTTP request as a new independent connection. HTTP cookies [4] are used to store the result of an initial authentication and represent afterwards the session state of an authenticated user. We will call such cookies *session cookies*. HTTP cookies can be set or modified with each HTTP response, where the web server includes an additional HTTP header `Set-Cookie` and the according cookie value. Additionally, cookies contain further session information such as `domain` and `path` to define the scope of a cookie (e.g., `docs.foo.com/accounts`) and a timestamp. Once a cookie is set, it is sent along with every HTTP request from the browser to the web server. Session cookies are a valuable target because they allow identity theft (see Section 7).

4. ADVERSARIAL MODEL

We assume that user agent (e.g., browser) and computer of the victim are not compromised. Furthermore, IdP and SP are benign.

For the attacks described in this paper we use the *web attacker model* [46]. By this means, the adversary only needs to register a domain and set up a malicious website. Visiting this malicious website can enforce the victim's browser to issue HTTP requests to the IdP. We assume an adversary able to lure the victim to a website controlled by him. This may be easily done through phishing [13] and malicious advertising [16]. Additionally, we assume that the victim is already authenticated to the IdP and therefore session cookies are set up.[2]

Our adversary has far fewer resources than the classical network-based attacker, who acts as man-in-the-middle (MITM) within the victim's communication. Since there is no need to read the network traffic, we assume that the user agent of the victim always communicates over encrypted TLS channels. Moreover, the victim only accepts communication partners with valid and trusted server certificates.

The goal of the adversary is to authenticate as victim to the attacked SP with stolen SAML assertions (Class 1: ACS Spoofing) or to authenticate as victim to the IdP with stolen session cookies (Class 2: XSS/UI redressing attacks).

[2]As the victim is using an IdP to reduce sign-on tasks, this is a very likely assumption.

5. METHODOLOGY

In this section we describe the selection criteria for the chosen cloud-based IdPs, introduce them, and explain the testing methodology.

5.1 Targeted Cloud IdPs

We selected six real-world SSO IdPs based on Wikipedia's comprehensive "SAML-based products and services" list [49] and Network World's review of eight prominent cloud-based SSO products [45].

As selection criteria, we applied the following conditions: (1) *On-demand.* We chose IdPs that are available as a Software-as-a-Service (SaaS), do not require installation, and configuration is done on the client's side. (2) *Widespread.* They must be widespread and are used by enterprises. (3) *Free accounts.* The IdPs offer free trail accounts with an activated SAML interface.

We strictly chose cloud-based SSO products and excluded on-premise software products like Shibboleth or OpenAM, that must be downloaded, installed, and configured. Depending on configuration, activated modules, and extensions the verification logic may vary. Therefore, a statement regarding the security of such frameworks, covering all existing deployments, cannot be made.

OneLogin. OneLogin [34] offers identity and access management as a cloud service for over 700 customers, including Netflix, Steelcase, Pandora, PBS, and has more than 12 millions of licensed users. OneLogin supports identity management features, user directory integration (e.g., Microsoft Active Directory), and various strong authentication methods (e.g., MFA and client certificate). The IdP supports SAML-based SSO with over 450 SPs, including Box, Concur, Google Apps, NetSuite, Salesforce, Workday, Yammer, and Zendesk.[3] Furthermore, it is important to remark that OneLogin also supports SSO with more than 3,600 websites by applying form-based authentication.

Okta. Okta [33] is a leading on-demand identity and access management service for enterprises with over 1,200 customers (e.g., London Gatwick Airport, Western Union, and LinkedIn). The IdP supports a large variety of SAML-based SPs, including Google Apps, Salesforce, Citrix GoToMeeting, and EchoSign. Okta also supports SSO with more than 3,000 websites by applying form-based authentication.

CloudSeal. Cloudseal [12] is a SAML-based IdP service offering MFA and support for a wide range of SPs (e.g., Salesforce and Google Apps). Additionally, Cloudseal provides a Java SDK to integrate SAML-based SSO into existing applications. At the time of this writing no official customer list is available.

SSOCircle. SSOCircle [43] is a free public SAML IdP that facilitates various strong authentication methods. The IdP supports SAML compliant SPs (e.g., Google Apps, GMail, Salesforce, and ServiceNow). We do not have any information about the number of current users.

WSO2 Stratos. WSO2 Stratos [50] is a cloud middleware that includes WSO2 Identity Server (IS) to provide security and identity management for web applications, services, and APIs. WSO2 IS supports SAML-based SSO and is also used as standalone product in large enterprises.

[3]`http://www.onelogin.com/blog/onelogin-adds-450th-saml-enabled-saas-vendor-to-sso-catalog/`

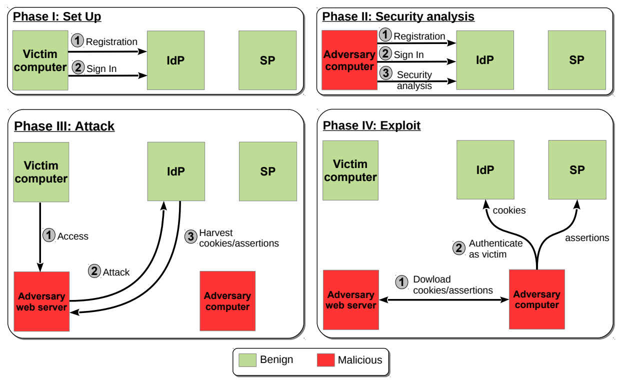


Figure 3: The four phases of our testing methodology assessment process.

Bitium. Bitium [6] is a SaaS IdP enabling identity management between different web applications via SAML-based SSO. It supports more than 1,750 different cloud applications like Google Apps, Salesforce, and NetSuite.

5.2 Testing Methodology

We had no access to the implementation code of the cloud-based SSO products. Therefore, we treated the IdPs as black boxes. In order to provide the security analysis and verify the results, we set up three strictly separated test systems:

- **Victim computer** used for registration on the target IdP, corresponding configuration, and SSO. In this manner, we simulated all actions a victim user would be able to perform.
- **Adversary web server** where the attack vectors (i.e. malicious website, iFrames, and AuthnRequests) were deployed. This server was accessible over the Internet. Additionally, a service on this server provides the harvested assertions and session cookies.
- **Adversary computer** used to send the stolen session cookies and assertions in order to authenticate as the victim.

For each IdP we conducted the following assessment process (see Figure 3):

Phase I: Set Up. (1) The victim registers and configures a victim account Vic_{Acc} from his victim computer. (2) Vic_{Acc} logs in on the IdP and uses the offered services.

Phase II: Security analysis. (1) The adversary registers and configures an adversary account Adv_{Acc} from his adversary computer. (2) Adv_{Acc} logs in on the IdP and evaluates the security of the application. (2) In order to find ACS Spoofing vulnerabilities, the adversary uses ACS Scanner. To discover Class 2: XSS and UI redressing flaws, he manually executes a penetration test. Finally, he creates the attack vectors and deploys them on the adversary web server.

Phase II: Attack. (1) The victim accesses the adversary web server from his victim computer. (2) By clicking on a link, the attacks discovered in Phase 2 will be executed. (3) The adversary web server harvests the assertions and session cookies.

Phase IV: Exploit. (1) In order to verify the success of the attack and to mitigate false positives results, the adversary downloads the cookies and/or assertions from the adversary web server to his adversary computer. (2) Thereafter, the cookies are used by the adversary to authenticate as Vic_{Acc} to the IdP and the assertions are used to authenticate as Vic_{Acc} to SP.

6. NOVEL ATTACK: ACS SPOOFING

This novel attack relies on a logical flaw in the IdP's SAML interface implementation, at the interplay between XML and HTTP, and allows to steal fresh and valid assertions. It affects *all* SPs having a trust relationship with the IdP.

6.1 Attack Description

Precondition. When the victim visits the malicious website A, the attack is carried out automatically. If the user is already authenticated to IdP, the attack executes in a fully transparent manner, without any further user interaction. Otherwise, the adversary may mask the malicious website A as a Service Provider $\widetilde{SP}$, thus the victim who has to authenticate to IdP believes that he has started an SSO protocol, which is in fact run with the accessed (malicious) $\widetilde{SP}$. Setting up a "Bad SP" is not harder than setting up an ordinary website, since no trust relationship with IdP must be established.

Attack. Figure 4 illustrates the detailed flow of the ACS Spoofing attack. It consists of the following steps:

1. **UA $\rightarrow$ A:** User U navigates its user agent UA to the malicious website A.

   Please note that the next two messages (step 2 and 3) are optional. They will only be executed if SP requires and validates the `InResponseTo` attribute of the assertion. For that purpose, A needs a valid `ID` for `<AuthnRequest>`, which was freshly generated by SP.
2. **A $\rightarrow$ SP:** A requests a protected resource URL_{SP} on an arbitrary SP resulting in a new SSO protocol run. This request can be made via a simple script call, e.g., curl[4], or a web browser.
3. **SP $\rightarrow$ A:** SP determines that no valid security context exists. Accordingly, SP issues an authentication request `<AuthnRequest(ID, SP,` ACS_{URL}`)>` and sends it Base64 encoded, along with URL_{SP}, as an HTTP 302 (redirect to IdP) to A. `ID` is a fresh random string and `SP` the identifier of the Service Provider. ACS_{URL}, included in the SP's `<AuthnRequest>`, specifies the endpoint to where the IdP must send the assertion.
4. **A $\rightarrow$ UA:** A generates a `<AuthnRequest>` and includes the $\overline{ACS_{URL}} = Bad_{URL}$, which is a URL controlled by the adversary. In the case that the `InResponseTo` attribute is validated by SP, A copies the `ID` of the `<AuthnRequest>` received in step 3, into its malicious `<AuthnRequest>`.
5. **UA $\rightarrow$ IdP:** Triggered by the HTTP redirect, a server-authenticated TLS connection is established between UA and IdP. UA uses the established TLS channel to transport `<AuthnRequest(ID, SP,` Bad_{URL}`)>`, along with URL_{SP}, to IdP.
6. **UA $\leftrightarrow$ IdP:** If the user is not yet authenticated, IdP identifies U by an arbitrary authentication mechanism (e.g., MFA).
7. **IdP $\rightarrow$ UA:** The signed assertion AA is embedded into a `<Response>` message and is sent Base64 encoded in an HTML form, along with URL_{SP}, to UA. According to the SAML standard, the IdP *must* use the $\overline{ACS_{URL}} = Bad_{URL}$ as HTTP POST destination ([8], Section 3.4.1).
8. **UA $\rightarrow$ A:** A JavaScript event in the HTML form triggers the HTTP POST of `<Response>` to Bad_{URL} (which is under the control of A).
9. **A $\rightarrow$ SP:** A can now impersonate U by submitting AA to SP.
10. **SP $\rightarrow$ A:** SP consumes AA, verifies the XML signature, and authenticates A as U. Therefore, SP grants A access to the protected service or resource, doing so by redirecting him to the originally accessed URL_{SP}.

6.2 Practical Evaluation

OneLogin. This cloud-based IdP is vulnerable to ACS Spoofing. Furthermore, it was possible to automate the attack. When the victim accessed the website A, the adversary opened for every supported SP a new iFrame carrying a malicious URL along with a self-generated `<Authn-Request>` targeted to OneLogin's SAML endpoint. Subsequently, the browser loaded each iFrame and launched multiple ACS Spoofing attacks in parallel. This attack variant enabled the adversary to steal assertions for *every* configured SAML SP with a *single* access to the malicious website on victim's part.

WSO2 Stratos. WSO2 Stratos is vulnerable to ACS Spoofing. Interestingly, when authenticating to WSO2 Stratos, the `<AuthnRequest>` did not contain any ACS_{URL} at all. By inserting a malicious $\overline{ACS_{URL}} = Bad_{URL}$, the adversary could set a new permanent default ACS_{URL} endpoint. Therefore, *one* spoofed `<AuthnRequest>` was suffice to automatically receive all assertions created in the following. This technique works for every federated SP.

SSOCircle and Bitium. We discovered that those IdPs do not verify the trustworthiness of the ACS_{URL}. Thus, ACS Spoofing was applicable.

6.3 ACS Scanner

In order to facilitate the security tests against ACS Spoofing we developed *ACS Scanner* – an automated penetration test tool available on the Internet.[5] Thus, ACS Scanner is platform independent and does not require any installation of additional software.

Configuration. There are three parameters, which can be configured in ACS Scanner:

- `IdP Endpoint URL` defines the URL where the attacked IdP is deployed. The malicious SAML Request will be sent to this URL.

- `SAML Request Issuer` defines the issuer of the SAML Request, e.g., `http://google.com`. In many cases, IdPs use this value for the generation of the assertion. Thus, ACS Scanner allows the change of the value.

[4] `http://curl.haxx.se/`

[5] `http://ssoattacks.org:8080/acsscanner/`

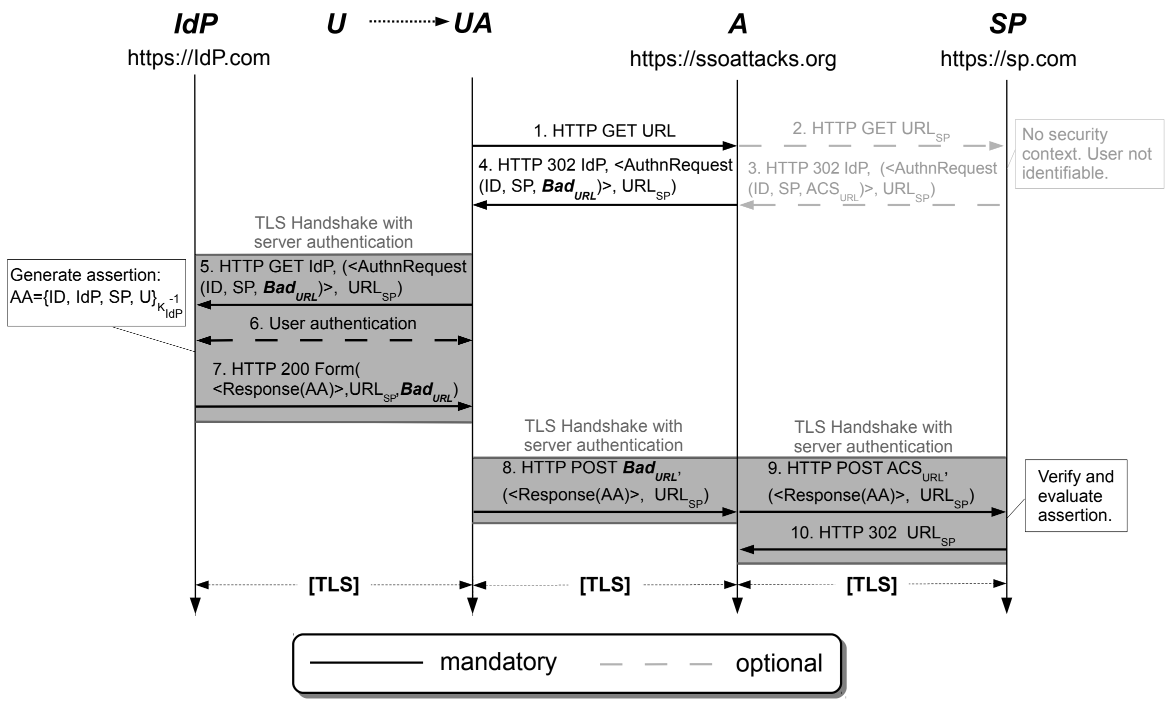

IdP
https://IdP.com
U
UA
A
https://ssoattacks.org
SP
https://sp.com
1. HTTP GET URL
2. HTTP GET URL_SP
No security context. User not identifiable.
4. HTTP 302 IdP, <AuthnRequest (ID, SP, Bad_URL)>, URL_SP)
3. HTTP 302 IdP, (<AuthnRequest (ID, SP, ACS_URL)>, URL_SP)
TLS Handshake with server authentication
Generate assertion: AA={ID, IdP, SP, U}K_IdP^-1
5. HTTP GET IdP, (<AuthnRequest (ID, SP, Bad_URL)>, URL_SP)
6. User authentication
7. HTTP 200 Form(<Response(AA)>,URL_SP,Bad_URL)
TLS Handshake with server authentication
TLS Handshake with server authentication
8. HTTP POST Bad_URL, (<Response(AA)>, URL_SP)
9. HTTP POST ACS_URL, (<Response(AA)>, URL_SP)
Verify and evaluate assertion.
10. HTTP 302 URL_SP
[TLS]
[TLS]
[TLS]
mandatory
optional

Figure 4: ACS Spoofing attack on standard SAML Web Browser SSO.

- `SAML Request` is an input field, where the malicious SAML authentication request can be configured manually. For normal operations this is not necessary, but experts have the option to build their own handcrafted request message. For example, when the SP verifies the `InResponseTo` attribute of the assertion, the pentester can use this field to provide a freshly created `<AuthnRequest>` from a federated SP. ACS Scanner provides a default SAML `<AuthnRequest>` as an example.

Evaluation. By clicking on the "Start Test" button ACS Scanner will start the security analysis:

1. ACS Scanner generates an unique `ID` (ID_1) and binds it to the malicious SAML Request. Then, both values are stored in the internal database.
2. ID_1 is attached to the ACS URL in the SAML Request, $ACS_{URL} = ssoattacks.org/acsscanner/ID_1$. In case that the tested IdP is vulnerable against ACS Spoofing, it will sent the token to ACS_{URL}.
3. ACS Scanner calls the URL of the IdP in a new browser tab. The user has to authenticate himself to the IdP if he is not already authenticated. Please note that ACS Scanner does not act as MITM and cannot intercept any credentials used on the IdP for the authentication.
4. If the IdP is vulnerable against ACS Spoofing, it will process the SAML Request and send the assertion to ACS_{URL}.
5. In case that ACS Scanner receives an assertion, it creates a relation between $ID_1 \leftrightarrow$ SAML Request $\leftrightarrow$ SAML Response. Based on this data, ACS Scanner classifies the according IdP as vulnerable and displays the results.

7. WEB ATTACKS AGAINST IDPS

Of the six IdPs tested, only Cloudseal and Okta were not vulnerable to ACS Spoofing, as they whitelist or ignore the ACS_{URL} parameter. However, we were able to compromise the security of both IdPs with a combination of XSS and UI redressing attacks. Our penetration test revealed that no IdP applies `X-Frame-Options` in HTTP response headers to mitigate framing attacks like clickjacking. While four IdPs (Okta, OneLogin, WSO2 Stratos, and Bitium) use the `secure` flag to protect session cookies during transit, only WSO2 Stratos and Bitium apply HTTP-Only cookies to prevent cookie theft through XSS attacks.

7.1 XSS and UI Redressing

A valid XSS attack allows to execute adversary-contributed script code within the web origin of a trusted site. To do so, generally a string is sent to the attacked site which will be embedded in the delivered webpage. In contrast to XSS an adversary needs trusted events like mouse clicks or key strokes to carry out UI redressing attacks. It is important to know that both attacks can be combined so that an adversary can use only XSS, only UI redressing or UI redressing and then XSS (or vice versa). We have operated with these techniques in our practical evaluation to attack the SSO systems.

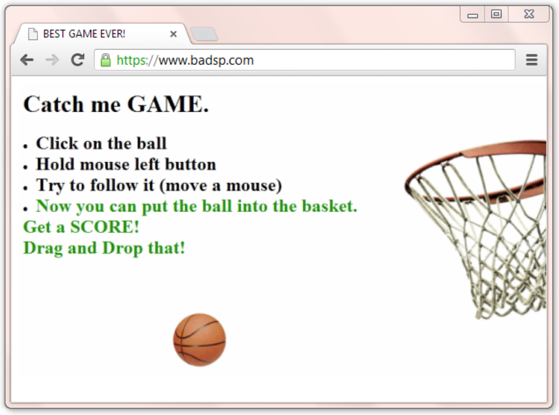
Please note that drag-and-drop attacks in combination with XSS are novel in the areas of SSO and IdPs. Though drag-and-drop attacks were introduced in April 2010 by Stone [44], they are still a critical issue. This statement will be underlined with XSS as it is described in the following practical evaluation.

7.2 Practical Evaluation

Okta. The attack we found requires a few specific clicks and a drag-and-drop event to insert a malicious XSS vector. Note that all these actions appear harmless to the victim. The combination allowed us to steal sensitive data, such as the IdP's session cookie. The victim, with a web browser like Firefox, has to perform the following steps (invisible to him) for a successful attack:

1. First, the victim has to visit a webpage controlled by the adversary. This webpage consists of three different elements: an iFrame rendered invisible by using the CSS property *opacity* and two elements for social engineering. In our proof of concept, these are two images: a ball and a basket (cf. Figure 5). The iFrame is loaded from the URL `https://foo.okta-admin.com/admin/settings/emails` – the wildcard `foo` should be replaced with the subdomain of the victim. At the time of the attack, the victim has to be logged in to Okta.
2. The Okta webpage inside the invisible iFrame has a *User Activation* button, which allows us to open a *View/Edit User Activation Email* window. We reach this window by triggering user's click on this invisible button, for example by asking the victim to press a "Start Game" button.
3. Inside this window, there is a form field where the user can type in the title. The information submitted via this title field is not filtered sufficiently by the server. Thus, we can inject JavaScript code, which will then be executed in the victim's browser (e.g., `<img src=x onerror=alert(document.cookie) x="`). However, the victim has to actively inject the JavaScript code; we cannot directly fire an HTTP POST request due to an existing CSRF [35] protection. This can be achieved by using the two pictures where the ball contains the malicious JavaScript code and where a drag of this ball injects this code into a dragable element like the *title* field.
4. The crucial point of the attack is to get the victim to inject the XSS vector of the adversary into the *title* field. In order to do so, the victim has to drag the ball into the basket. In our case, the ball has the HTML5 event handler *ondragstart* with our XSS vector as its data. Upon the event of dragging the ball into the basket, the vector will be automatically dropped into the title field, for the reasons of the basket being placed exactly over it.
5. The victim has to submit the form with the XSS vector inside its title field by clicking on the *Submit* button. This can be done with an element like the moving basket. To compel the victim to clicking on the basket, a game score that increases with each click may be introduced.
6. When the form is submitted, the malicious code will be automatically saved and executed. This allows the adversary to retrieve the session cookie of the IdP stored in the browser, and thus impersonate the victim (e.g., as an administrator). As IdP administrator the adversary is able to compromise the security of the whole Okta service. Please note that we also have a stored XSS vulnerability here.

OneLogin. The XSS flaw in `https://app.onelogin.com` can be easily exploited due to the fact that there is no CSRF protection. A simple HTTP GET request triggered by the browser (e.g., by image loading) is sufficient.



Catch me GAME.

- Click on the ball
- Hold mouse left button
- Try to follow it (move a mouse)
- Now you can put the ball into the basket.

Get a SCORE!
Drag and Drop that!

Figure 5: Combined XSS/UI redressing attack on Okta. The victim clicks on the ball, drags it to the basket, and drops it over the basket. By doing this, the session cookie will be hijacked by executing the inserted XSS code.

Cloudseal. We found one persistent XSS that allowed us to steal the session cookies with a combined XSS/UI redressing attack.

SSOCircle. Out of the four XSS vulnerabilities we found in SSOCircle, two could be exploited by HTTP GET requests. As in the case of OneLogin, no CSRF protection was deployed.

WSO2 Stratos. Five out of seven XSS vulnerabilities we found were persistent and feasible through the use of HTTP requests. WSO2 Stratos deploys HTTP-Only cookies. Therefore, we were not able to steal the session cookies. Nevertheless, these findings are severe, as there are many other ways to exploit XSS, aside from cookie theft. The Browser Exploitation Framework[6] arrestingly demonstrates several methods (e.g., capturing and transmitting user's keystrokes).

Bitium. We found several XSS vulnerabilities in Bitium but cookie theft was not possible as HTTP-Only cookies were deployed.

8. COUNTERMEASURES

This section provides countermeasures against ACS spoofing and cookie theft.

8.1 ACS Spoofing Countermeasures

We review four countermeasures against ACS Spoofing, along with known weaknesses.

Whitelisting. One way to mitigate ACS Spoofing is to use a whitelist of allowed ACS_{URL} values for each and every SP, stored at IdP. This may induce a significant management overhead for large IdPs. The exchange of ACS_{URL} values could be done with SAML metadata which is used to establish federations.

Signing Authentication Requests. In theory, signing authentication requests would make the injection of a malicious ACS_{URL} impossible. According to [8, Section 3.4.1] the `<AuthnRequest>` *should* be signed. However, our inves-

[6]`http://beefproject.com/`

tigation shows that this is usually not the case in real-world implementations. Only one (Cloudseal) out of six evaluated IdPs uses `<AuthnRequest>` signing. Additionally, [8] states that the ACS_{URL} of a signed `<AuthnRequest>` is always a trusted destination. This opens another interesting attack vector. Interestingly, WSO2 Stratos chose to implement `<AuthnRequest>` signing to mitigate ACS Spoofing. Our observations revealed that the XML Signature verification module of WSO2 Stratos was susceptible to XML Signature wrapping attacks (in the line of [42]), which renders the integrity protection of XML signatures useless. Therefore, ACS Spoofing attacks may be possible even if signed `<AuthnRequest>` messages are used. Furthermore signing authentication requests may require implementation changes of IdPs and increases the management overhead (registering public keys at the IdP).

Recipient Attribute Evaluation. To counter ACS Spoofing, the IdP can embed the value of the ACS_{URL} into the issued assertion as `Recipient` attribute in the `<SubjectConfirmationData>` element. Only if the `Recipient` value is equal to the SP's own ACS endpoint, the assertion can be considered valid. Unfortunately, [8, Section 2.4.1.2] mandates this attribute as *optional*. Therefore, only few of the SPs evaluate it.

OASIS HoK Profile. An effective way to mitigate ACS Spoofing is using the OASIS HoK Profile [28], which is standardized but rarely used. The user needs a TLS client certificate as a prerequisite but any self-signed certificate is sufficient, as neither the IdP nor the SP is required to validate its trust chain. A self-signed certificate could easily be created and automatically imported into the browser through interaction with a small OpenSSL CA located at the IdP. This approach is transparent for the user. All major browsers support client certificates. Firefox, IE, and Chrome can be configured to automatically select a certificate if a web server requests one. Then, no user interaction is needed anymore. The issued assertion is cryptographically bound to this client certificate by including either the certificate itself or a hash of it in the signed assertion.

Regarding the standard SSO protocol flow in Figure 1, the client certificate $Cert_{UA}$ is sent to the IdP in the TLS handshake (step 4). The UA proves possession of the private key belonging to the client certificate by successfully completing the handshake. Upon authentication, the IdP creates the authentication assertion AA which additionally includes $Cert_{UA}$ and is signed with the IdP's private key K_{IdP}^{-1} (step 5). AA is only valid if it is sent to the SP over a mutually authenticated TLS channel established by the UA with the same client certificate. This approach does not prevent assertion theft (e.g., by ACS Spoofing). However, stolen authentication tokens are worthless since they are cryptographically bound to the legitimate browser being in possession of the certificate's private key.

The cryptographic binding of the assertion to the TLS client certificate protects against a wide range of spoofing (including ACS Spoofing) and MITM attacks during the SSO protocol run.

8.2 Cookie Theft Countermeasures

In this subsection, we review best-practice countermeasures against cookie theft, along with known weaknesses. In summary, all of these countermeasures can be bypassed in several ways.

Enforcing Secure Transport. To mitigate cookie theft effectuated via eavesdropping on the network traffic, the cookies are sent over TLS connections. This policy is enforced by setting the cookie's `secure` flag in the `Set-Cookie` HTTP response header. Since the rise of comfortable packet sniffers for cookie theft (e.g., Firesheep [7]), this option is prevalently used. However, cookie theft via XSS is still possible because cookies are only protected during transit.

XSS Filtering. XSS attacks can be mitigated by server- and client-side filtering. In practice, server-side defense of XSS is primarily used. However, Nadji et al. [32] have shown, that server-side filtering, as stand-alone countermeasure, is insufficient. Furthermore, Bates et al. [5] have analyzed leading client-side XSS filters and have found severe security flaws in them. Even a combination of both techniques cannot prevent all XSS attacks as Heiderich has shown [24].

HTTP-Only Cookies. A simple and effective way to prevent cookie theft through XSS is to use the `HTTP-Only` flag. In this case, access of client-side scripts to these cookies is blocked by the browser. However, other attack techniques such as cross-site tracing [30] and using XMLHttpRequests [37] can be employed, allowing an adversary to steal HTTP-Only cookies. In addition, HTTP-Only cookies are not widely deployed and may disrupt a webpage's functionality [53].

Cryptographic Cookie Protocols. Some ineffective efforts to secure cookies by deploying public key-based authentication mechanisms have taken place [38, 18]. The proposed cookie protocols guarantee authentication, confidentiality, and integrity. However, neither signing nor encrypting cookies does deter an adversary from transferring a cookie from one browser to another.

Anti-CSRF Tokens. To forbid the processing of malicious server side HTTP requests to do actions like cookie theft one can use anti-CSRF tokens. In this scenario, the server checks in the current HTTP request, if the user or rather the victim is sending a CSRF token, in form of a not guessable random string, generated by the previous HTTP request. If so, the server will accept the request; otherwise it will be rejected. However, this countermeasure does not work in the case of UI redressing attacks due to the reason that the victim will be lured to send a valid token to the server.

X-Frame-Options. To mitigate framing attacks like clickjacking X-Frame-Options have been proposed. By sending an additional X-Frame-Options header in the HTTP response message, a website can instruct a browser not to render the content of the webpage inside an iFrame. This mitigates many UI redressing attacks. X-Frame-Options are supported by all major browsers. However, Heiderich has shown that this defense can be bypassed via Java applets or LiveConnect [24].

Phishing. To solve the problem of, for example, being clickjacked in general, phishing attacks should be prohibited. Although this problem is obvious, it is still not solved in practice. Dhamija et al. [13] showed in 2006 that 23% of the users do not look into cues such as the address or status bar. Making it nearly impossible for an attacker to lure the user to a malicious website is thus a heavy task. However, one way to address it could be to optimize the UI of future browser versions for a better visibility of warning messages.

8.3 Preventing ACS Spoofing and Cookie Theft

We present a countermeasure that simultaneously protects an SSO system against ACS Spoofing and cookie theft. It combines the OASIS HoK Profile [28] for initial authentication with a strong binding of session cookies to the TLS channel (along the lines of [15]) for repeated authentication.

Combining HoK with Cookie Binding. The countermeasures considered so far are used to strengthen the SSO authentication process by securing the generated token against spoofing and MITM attacks. However, these mechanisms do not grant any further protection for the HTTP session cookies used to authenticate the user against the IdP or SP afterwards.

Our investigation shows that even if one has a secure SSO protocol in place, one small flaw in a web application can break the whole SSO system (cf. Section 7). Therefore, we propose to extend the usage of the TLS client certificate applied in the HoK Profile to HTTP session cookies.

According to the ideas introduced in [15], an unforgeable fusion between the client certificate and the cookie can be done as follows:

$$C_{bound} := v \mid\mid HMAC_k(v \mid\mid Cert_{UA}),$$

where v is the value of a standard HTTP cookie, $Cert_{UA}$ is the client certificate, and HMAC is a message authentication code computed over v and $Cert_{UA}$ with the key k ($\mid\mid$ denotes a string concatenation). In this manner, if the cookie gets stolen, it can be used only through a TLS channel authenticated with $Cert_{UA}$, which in turn is only possible for the party who knows the private key of the client's certificate.

However, this technique of strengthening cookie authentication process leads to further requirements being imposed on the service responsible for the HTTP session management. Namely, it must have access to the client certificate applied during the TLS handshake, causing the need for the service being extended.

In order to demonstrate the feasibility of the combined countermeasure, we implemented the cookie binding in the popular open source framework SimpleSAMLphp [47]. We used HMAC-SHA256 as the keyed hash message authentication code function for the cookie binding and conducted a performance evaluation. Compared to the standard cookie authentication, our cookie binding was only 12,1% slower. Further details are presented in Appendix A.

9. CONCLUSION

Developing a secure SSO solution is a nontrivial task. Our findings show that vulnerabilities in actual SAML-based cloud IdPs can be severely exploited, leading to a complete failure in regards to the security of the IdP and all SPs.

Due to the fact that SAML is a very flexible and extensible standard, the corresponding specifications are complex and distributed over a bulk of documents. Developers can get lost in the specification and may overlook important security-relevant constraints. This can result in vulnerable implementations, as the discovered ACS Spoofing attack demonstrates. Nevertheless, SAML is a matured and well-designed standard. Throughout the specification, multiple security recommendations are given for the purpose of avoiding common pitfalls. Still, this does not guarantee the absence of flaws in real-world implementations.

Even if the SSO protocol is considered secure, the prevalent cookie-based client authentication creates an attack-surface sufficient for identity theft done through XSS/UI redressing. Our results confirm the significance of these attacks for the security in SSO systems.

In order to fix the mentioned problems, we analyzed several countermeasures. Our preferred mitigation is the OASIS HoK Profile with cookie binding combining the ease of SSO with a cryptographically strengthened client authentication. Our solution hardens both the SSO protocol and the session cookies by establishing mutually authenticated channels between the browser and the other participating entities (i.e. IdP and SP). This builds a holistic authentication layer that prevents a wide range of attacks, including MITM, ACS Spoofing, and XSS/UI redressing vulnerabilities. The practical feasibility of our approach is shown by a proof-of-concept implementation in the open source framework SimpleSAMLphp. The accompanied performance analysis (cf. Appendix A) demonstrates that the proposed cookie binding performs well. No changes of browser, web server, and TLS protocol are necessary. Finally, our ideas are generic and can directly be applied to other SSO protocols (e.g., OAuth or OpenID).

In future, we plan to extend the functionality of ACS Scanner in order to cover attacks on signed SAML Requests (e.g., signature exclusion and XML Signature wrapping) that bypass the protection of the digital signature.

10. REFERENCES

[1] Apache Software Foundation. JMeter Project. `http://jmeter.apache.org`.

[2] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, G. Pellegrino, and A. Sorniotti. From Multiple Credentials to Browser-Based Single Sign-On: Are We More Secure? In *SEC*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 68–79. Springer, 2011.

[3] A. Armando, R. Carbone, L. Compagna, J. Cuéllar, and M. L. Tobarra. Formal Analysis of SAML 2.0 Web Browser Single Sign-On: Breaking the SAML-based Single Sign-On for Google Apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008*, pages 1–10, Alexandria and VA and USA, 2008. ACM.

[4] A. Barth. HTTP State Management Mechanism. RFC 6265 (Proposed Standard), Apr. 2011.

[5] D. Bates, A. Barth, and C. Jackson. Regular expressions considered harmful in client-side XSS filters. In *Proceedings of the 19th international conference on World wide web*, WWW '10, pages 91–100, New York, NY, USA, 2010. ACM.

[6] Bitium Inc. Bitium. `https://www.bitium.com/`, 2014.

[7] E. Butler. Firesheep, 2010. `http://codebutler.com/firesheep/`.

[8] Cantor, S. et al. Assertions and Protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. `http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf`, Mar. 2005.

[9] Cantor, S. et al. Profiles for the OASIS Security Assertion Markup Language (SAML) V2.0. OASIS Standard, 15.03.2005, Mar. 2005. `http://docs.oasis-open.org/security/saml/v2.0/saml-profiles-2.0-os.pdf`.

[10] Carnegie Mellon University. CERT Coordination Center, 1995–2014. `http://www.cert.org`.

[11] CERT. Advisory CA-2000-02 Malicious HTML Tags Embedded in Client Web Requests, 2000. http://www.cert.org/advisories/CA-2000-02.html.

[12] Cloudseal OU. Cloudseal. http://www.cloudseal.com/, 2011–2014.

[13] R. Dhamija, J. D. Tygar, and M. Hearst. Why phishing works. In *SIGCHI Conference on Human Factors in Computing Systems*, CHI '06, pages 581–590, New York, NY, USA, 2006. ACM.

[14] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[15] M. Dietz, A. Czeskis, D. Balfanz, and D. S. Wallach. Origin-Bound Certificates: A Fresh Approach to Strong Client Authentication for the Web. In *21st USENIX Security Symposium*, Bellevue, WA, Aug. 2012.

[16] M. Finifter, J. Weinberger, and A. Barth. Preventing capability leaks in secure javascript subsets. In *NDSS*, 2010.

[17] D. Florencio and C. Herley. A large-scale study of web password habits. In *Proceedings of the 16th international conference on World Wide Web*, WWW '07, pages 657–666, New York, NY, USA, 2007. ACM.

[18] K. Fu, E. Sit, K. Smith, and N. Feamster. Dos and Don'ts of Client Authentication on the Web. In *10th USENIX Security Symposium*, Washington D.C., 2001.

[19] S. Gajek, T. Jager, M. Manulis, and J. Schwenk. A browser-based Kerberos authentication scheme. In S. Jajodia and J. López, editors, *Computer Security - ESORICS 2008, 13th European Symposium on Research in Computer Security, Málaga, Spain, October 6-8, 2008. Proceedings*, volume 5283 of *Lecture Notes in Computer Science*, pages 115–129. Springer, August 2008.

[20] T. Groß. Security analysis of the SAML Single Sign-on Browser/Artifact profile. In *Annual Computer Security Applications Conference*. IEEE Computer Society, 2003.

[21] T. Groß and B. Pfitzmann. SAML artifact information flow revisited. Research Report RZ 3643 (99653), IBM Research, 2006. http://www.zurich.ibm.com/security/publications/2006.html.

[22] Guangdong, B. et al. AUTHSCAN: Automatic Extraction of Web Authentication Protocols from Implementations. In *NDSS*, 2013.

[23] R. Hansen and J. Grossman. Clickjacking, 2008. http://www.sectheory.com/clickjacking.htm.

[24] M. Heiderich. *Towards Elimination of XSS Attacks with a Trusted and Capability Controlled DOM*. PhD thesis, Ruhr-University Bochum, Bochum, May 2012.

[25] M. Heiderich, M. Niemietz, F. Schuster, T. Holz, and J. Schwenk. Scriptless attacks: stealing the pie without touching the sill. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 760–771, New York, NY, USA, 2012. ACM.

[26] M. Heiderich, J. Schwenk, T. Frosch, J. Magazinius, and E. Z. Yang. mxss attacks: Attacking well-secured web-applications by using innerhtml mutations. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 777–788, New York, NY, USA, 2013. ACM.

[27] C. Karlof, U. Shankar, J. D. Tygar, and D. Wagner. Dynamic Pharming Attacks and Locked Same-Origin Policies for Web Browsers. In *14th ACM conference on Computer and communications security (CCS)*, pages 58–71, New York, NY, USA, 2007. ACM.

[28] N. Klingenstein. SAML V2.0 Holder-of-Key Web Browser SSO Profile. OASIS Committee Draft 02, 05.07.2009, 2009. http://www.oasis-open.org/committees/download.php/33239/sstc-saml-holder-of-key-browser-sso-cd-02.pdf.

[29] D. Kormann and A. Rubin. Risks of the Passport single signon protocol. *Computer Networks*, 33(1–6):51–58, 2000.

[30] A. Manion. Vulnerability Note VU#867593, 2003. http://www.kb.cert.org/vuls/id/867593.

[31] C. Masone, K.-H. Baek, and S. Smith. WSKE: Web Server Key Enabled Cookies. In S. Dietrich and R. Dhamija, editors, *Financial Cryptography*, volume 4886 of *Lecture Notes in Computer Science*, pages 294–306. Springer, 2007.

[32] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *NDSS*, 2009.

[33] Okta Inc. Okta. http://www.okta.com/, 2014.

[34] OneLogin Inc. OneLogin. http://www.onelogin.com/, 2010–2014.

[35] OWASP Foundation. Cross-Site Request Forgery (CSRF). https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF), 2013.

[36] OWASP Foundation. OWASP Top 10 - 2013: The Ten Most Critical Web Application Security Risks. https://www.owasp.org/index.php/Top_10_2013, 2013.

[37] W. Palant. (CVE-2009-0357) XMLHttpRequest allows reading HTTPOnly cookies, 2007. https://bugzilla.mozilla.org/show_bug.cgi?id=380418.

[38] J. S. Park and R. S. Sandhu. Secure Cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, 2000.

[39] B. Pfitzmann and M. Waidner. Analysis of Liberty Single-Sign-on with Enabled Clients. *IEEE Internet Computing*, 7(6):38–44, 2003.

[40] J. Ruderman. Bug 154957 – iframe content background defaults to transparent, 2002. https://bugzilla.mozilla.org/show_bug.cgi?id=154957.

[41] M. Slemko. Microsoft Passport to Trouble, 2001. http://www.znep.com/~marcs/passport/.

[42] J. Somorovsky, A. Mayer, J. Schwenk, M. Kampmann, and M. Jensen. On breaking saml: Be whoever you want to be. In *21st USENIX Security Symposium*, Bellevue, WA, Aug. 2012.

[43] SSOCircle. SSOCircle. http://www.ssocircle.com/, 2007–2014.

[44] P. Stone. Next Generation Clickjacking. http://www.contextis.co.uk/documents/5/Context-Clickjacking_white_paper.pdf, April 2010.

[45] D. Strom. Single sign-on moves to the cloud. http://www.networkworld.com/article/2161919/access-control/single-sign-on-moves-to-the-cloud.html, 2012.

[46] S.-T. Sun and K. Beznosov. The Devil is in the (Implementation) Details: An Empirical Analysis of OAuth SSO Systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 378–390, New York, NY, USA, 2012.

[47] Uninett AS. SimpleSAMLphp Project, 2014. `http://www.simplesamlphp.org`.

[48] R. Wang, S. Chen, and X. Wang. Signing Me onto Your Accounts through Facebook and Google: a Traffic-Guided Security Study of Commercially Deployed Single-Sign-On Web Services. In IEEE, editor, *Security & Privacy 2012*, 2012.

[49] Wikipedia. Saml-based products and services. `http://en.wikipedia.org/wiki/SAML-based_products_and_services`, March 2014.

[50] WSO2 Inc. WSO2 StratosLive. `https://stratoslive.wso2.com/`, 2014.

[51] L. Xing, Y. Chen, X. Wang, and S. Chen. Integuard: Toward automatic protection of third-party web service integrations. In *NDSS*, 2013.

[52] D. E. Yuchen Zhou. Automated testing of web applications for single sign-on vulnerabilities. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, Aug. 2014. USENIX Association.

[53] Y. Zhou and D. Evans. Why Aren't HTTP-only Cookies More Widely Deployed? In *Web 2.0 Security and Privacy 2010 (W2SP)*, 2010.

[54] G. Zuchlinski. The Anatomy of Cross Site Scripting. *Hitchhiker's World*, 8, 2003.

APPENDIX

A. IMPLEMENTATION & PERFORMANCE

In order to demonstrate the feasibility of the combined countermeasure (cf. Section 8.3), we implemented the cookie binding in SimpleSAMLphp (SSP) [47]. SSP already supports OASIS HoK (based on the previous work of one of the co-authors) and thus had to be extended only by the cookie binding. Additionally, we present a performance evaluation of our cookie binding implementation.

A.1 Cookie Binding

We introduced a new `session.cb` configuration parameter to enable or disable cookie binding. Furthermore, we created two new class methods. The `createBC($rnd, $cert, $key)` method creates a cookie bound to the TLS client certificate, where `$rnd` is an arbitrary value (e.g., a random session ID), `$cert` is the Base64 encoded client certificate of the UA, and `$key` specifies the HMAC secret key k. We applied the standard PHP 5 HMAC function `hash_mac()` which supports several hashing algorithms. The simplified source code of the class method is:

```
function createBC($rnd, $cert, $key) {
  $data = $rnd . $cert;
  $appendix = hash_hmac('sha256', $data, $key);
  $ret = $rnd . '_' . $appendix;
  return $ret;
}
```

To verify the authenticity, integrity, and binding to the TLS client certificate of the received cookies, `verifyBC($cookie, $rnd, $cert, $key)` was introduced, where `$cookie` defines the value of the bound session cookie. All other input parameters have the same purpose as in the `createBC()` method. The simplified source code is shown in the following:

```
function verifyBC($cookie, $rnd,$cert, $key) {
  $nc = createBC($rnd, $cert, $key);
  if ($nc === $cookie) { return TRUE; }
  else { return FALSE; }
}
```

To mitigate downgrade attacks, where an adversary cuts off the HMAC value from the cookie, the usage of cookie binding is enforced by the `session.cb` configuration parameter. These SSP modifications required 97 modified or added lines in the SSP source code.

A.2 Cookie Binding Performance

Cookie-based authentication is a performance-critical issue in every web application. Therefore, we conducted a performance evaluation of our cookie binding implementation, reporting our findings below.

Test environment All experiments were performed against an Apache 2.2.2 web server running on a Windows Vista system with a 3.0 GHz Core 2 Duo CPU and 2 GB of RAM. The server and the client were connected to a dedicated Gigabit link with a 0.3 ms roundtrip time. All performance tests were conducted with Apache JMeter 2.9 [1].

Analysis In order to demonstrate that the performance impact of adding cookie binding to web applications is minimal, we have evaluated our SSP implementation. We considered three different test cases using a special crafted webpage including the SSP framework:

1. *Unauthenticated requests.* In order to provide a baseline for comparison, the webpage is loaded without providing any authentication cookie. Therefore, no authenticated user session is established.
2. *Authentication with cookies.* The client sends a valid SSP authentication cookie to the webpage.
3. *Authentication with cookie binding.* The client sends a valid SSP authentication cookie bound to a TLS client certificate to the webpage which triggers the cookie binding verification.

For each case, we devised a separate test plan in Apache JMeter and made 25,000 successive requests to the webpage using TLS. Test cases 1 and 2 facilitated server-authenticated channels, while test case 3 dealt with mutually authenticated TLS channels. Additionally, we ensured that for each test case the HTTP response message had the same size. We used HMAC-SHA256 as the keyed hash message authentication code function for the cookie binding. The results are shown in Table 2. When compared to the standard cookie authentication, our cookie binding implementation was only 12,1% slower.

Table 2: Performance evaluation results.

	Latency (ms)			
Test case	**Avg**	**Median**	**Min**	**Max**
1. Unauthenticated	22.48	18	14	230
2. Standard cookie	27.19	27	17	228
3. Bound cookie	30.47	30	21	525