Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest & Audit-Report SimpleSAMLphp 11.2017

Cure53, Dr.-Ing. M. Heiderich, MSc. N. Krein, BSc. T.-C. Hong, BSc. D. Weißer, MSc. G. Kopf, MSc. N. Kobeissi, BSc. J. Hector

Index

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Introduction

"SimpleSAMLphp is an award-winning application written in native PHP that deals with authentication. The project is led by UNINETT, has a large user base, a helpful user community and a large set of external contributors."

From https://simplesamlphp.org/

This report documents the findings of a penetration test and security code audit targeting the SimpleSAMLphp. The project was completed by Cure53 in cooperation with one expert from Secfault Security GmbH in November 2017. It yielded ten security-relevant findings.

Besides the aforementioned invited Secfault Security tester, six members of the Cure53 project were also involved in this project. The entire team of seven testers was allocated a time budget of eighteen days to investigate the target. It must be stated that the tests were sponsored by Mozilla via the Secure Open Source programme. To facilitate communications, the Cure53 team was in an ongoing email contact with Jaime Perez Crespo who is the SimpleSAMLphp project's maintainer.

Several components were placed in scope of this assessment by the SimpleSAMLphp client. Specifically encompassed by this test were, first and foremost, the SimpleSAMLphp application, as well as the SimpleSAMLphp low-level SAML2 library, and the SimpleSAMLphp module installer. Last but not least, the testers also analyzed the *xmlseclibs* library which is heavily used in the project. In addition, several SimpleSAMLphp modules which underpin the overall use of the product were also specified as needing attention. In fact, prior to the beginning of the test, the SimpleSAMLphp maintainer delivered a list of modules that Cure53 was supposed to audit. A further very detailed documentation pertaining to the scope was supplied to Cure53 and enumerated all code parts that were deemed as calling for security review and checkup.

A range of methods and approaches was employed during this assignment. In addition to a cloud VM that Cure53 put in place to enable testing, the SimpleSAMLphp team also made their deployment available to testers. For the latter environment, Cure53 had access to an Identity Provider and a Service Provider to test with. This was done to make sure that the test setup is as close as possible to a real-life deployment.

As the tests were moving forward, an action plan was devised and split the work into five Work Packages. It was agreed that WP1 will entail general PHP code audit, WP2 cover tests against SAML IdP implementation, and WP3 was tasked with identification and

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

auditing of crypto-relevant parts. The remaining two components were WP4 with tests against commonly used modules and WP5 comprising tests against *xmlseclibs*.

In the following sections the report will first provide more technical details on the scope. It then lists and discusses all findings, finally presenting a conclusion on the overall security level of the SimpleSAMLphp code and other scope objects as perceived by Cure53 upon the completion of this project.

Scope

- **SimpleSAMLphp, SAML2 and *xmlseclibs* Codebase**
  - https://github.com/simplesamlphp/simplesamlphp
  - https://github.com/simplesamlphp/saml2
  - https://github.com/robrichards/xmlseclibs
  - https://github.com/simplesamlphp/composer-module-installer
- **An IdP was made available for this assessment**
  - https://ssp-demo-idp.paas2.uninett.no/simplesaml/
- **An SP was made available for this assessment**
  - https://ssp-demo-sp.paas2.uninett.no/simplesaml/

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *SSP-01-001*) for the purpose of facilitating any future follow-up correspondence.

SSP-01-001 XMLSEC: Casting Cryptographic Keys / Signature Bypass *(**Critical**)*

The *XMLSecurityKey* class defined in *xmlseclibs/src/XMLSecurityKey.php* specifies a generic object for holding cryptographic key material for various algorithms. Two instances of this class could hold two different types of cryptographic keys, for instance symmetric and asymmetric keys. The class furthermore defines a function named *verifySignature*, which is used to verify asymmetric signatures by employing the respective object's cryptographic key.

Affected File:
xmlseclibs/src/XMLSecurityKey.php

Affected Source:

```
public function verifySignature($data, $signature)
{
    switch ($this->cryptParams['library']) {
              case 'openssl':
              return $this->verifyOpenSSL($data, $signature);
              case (self::HMAC_SHA1):
              $expectedSignature = hash_hmac("sha1", $data, $this->key, true);
              return strcmp($signature, $expectedSignature) == 0;
    }
}
```

It should be noted that this function does not actually check the type of key it is operating on. This is generally not recommended as it makes it easy to confuse different types of cryptographic keys. A more advisable design would be to introduce distinct types for different keys. One particular key type supported by *XMLSecurityKey* concerns HMAC keys. When operating on an HMAC key, the *XMLSecurityKey* class will simply use its internal key material as an HMAC secret. The *verifySignature* function would compare the "*signature*" that is to be verified with an HMAC over the "*signed*" data.

The *verifySignature* function is used in various parts of other libraries. One example is provided in the code below.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected File:
saml2/src/SAML2/HTTPRedirect.php

Affected Source:

```
public static function validateSignature(array $data, XMLSecurityKey $key)
{
[...]

    if ($key->type !== XMLSecurityKey::RSA_SHA1) {
        throw new \Exception('Invalid key type for validating signature on query
string.');
    }
    if ($key->type !== $sigAlg) {
        $key = Utils::castKey($key, $sigAlg);
    }

    if (!$key->verifySignature($query, $signature)) {
        throw new \Exception('Unable to validate signature on query string.');
    }
}
```

It can be observed that this function will inspect the type of the signature to be validated. If it is not of the same type as the provided key, the key will be "casted" to the type of the signature. This means that if a SAML document contains a "*signature*" of the type "HMAC", then the *RSA_SHA1* key will be casted to the HMAC type as well. In this case, the PEM-encoded RSA public key will be used as a secret key for computing an HMAC over the data to be "*signed*".

It should be assumed that an attacker has access to the RSA public key used for verification. Hence, an attacker could craft a SAML document containing an HMAC "*signature*" which would be accepted by the analyzed implementation. This would hold even though the attacker did not have access to the private *signing* key. The sequence enables remote unauthenticated attackers to bypass the SAML signature validation function. It should however be noted that the feasibility of this attack depends on how the *verifySignature* function is invoked. Invoking it in a way where a condition is *($objXMLSecDSig->verify($key) !== 1)* effectively prevents the issue. Further clarification is needed to highlight that there is another option for bypassing the SAML signature scheme in this setting. Specifically, it entails making the *verifySignature* function return the value of *-1*. This could, for instance, be achieved by having the OpenSSL signature verification code fail (e.g., by supplying an overly long signature).

There are several possible ways for addressing this issue. As a first recommendation, the *return* value of the *verifySignature* function should always be checked against its type and the *0* value. Moreover, using one common class for holding multiple types of

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

cryptographic keys is generally a bad practice and should be discouraged. If possible, different classes for individual cryptographic algorithms should be introduced. Casting cryptographic keys to other types should also be avoided. It must be emphasized that the latter is the core issue of the described vulnerability as far as the auditors are concerned.

SSP-01-002 XMLSEC: Various XPath Injections *(Medium)*

The analyzed solution makes use of *XPath* for querying XML documents. This is required, for instance, when verifying the references of an XML signature in a SAML document.

Affected File:
pxmlseclibs/src/XMLSecurityDSig.php

Affected Source:

```
if ($uri = $refNode->getAttribute("URI")) {
	$arUrl = parse_url($uri);
	if (empty($arUrl['path'])) {
	if ($identifier = $arUrl['fragment']) {

		/* This reference identifies a node with the given id by using
		 * a URI on the form "#identifier". This should not
			include comments.
		*/
		$includeCommentNodes = false;

		$xPath = new DOMXPath($refNode->ownerDocument);
		if ($this->idNS && is_array($this->idNS)) {
		foreach ($this->idNS AS $nspf => $ns) {
			$xPath->registerNamespace($nspf, $ns);
		}
		}
		$iDlist = '@Id="'.$identifier.'"';
		if (is_array($this->idKeys)) {
		foreach ($this->idKeys AS $idKey) {
			$iDlist .= " or @$idKey='$identifier'";
		}
		}
		$query = '//*['.$iDlist.']';
		$dataObject = $xPath->query($query)->item(0);
```

It can be observed that the affected *processRefNode* function resolves internal XML references by turning them into *XPath* expressions. An attacker could therefore inject

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

their own *XPath* sub-expressions into the *$query* expression. One example of this kind of an attack would be to use a reference with a name like *#"] | //**, which would change the expression in use. During this assessment no way for exploiting this issue in order to bypass the SAML signature validation could be identified. Nevertheless, *XPath* Injections comprise a well-known problem and could potentially have other negative consequences for the targeted application, including DoS issues or rejecting actually valid SAML messages.

Please note that the above code excerpt is only an example. The problem appears to be a systemic issue within the entire code-base: all *XPath* queries using data from the XML document appear to be affected. The recommended fix is to apply a strict whitelist under the premise of a revised filtering approach for XML data that is part of the *XPath* expressions. Ideally the allowed character set should be restricted to the minimal size required. Special characters - like quotes - should generally be forbidden.

SSP-01-003 XMLSEC: DoS in staticLocateKeyInfo *(Medium)*

The analyzed code contains a Denial of Service condition (DoS) in the *staticLocateKeyInfo* function. Code below can be observed to understand the problem at hand.

Affected File:
xmlseclibs/src/XMLSecEnc.php

Affected Source:

```
public static function staticLocateKeyInfo($objBaseKey=null, $node=null)
{
	if (empty($node) || (! $node instanceof DOMNode)) {
	return null;
	}
	[...]
	foreach ($encmeth->childNodes AS $child) {
	switch ($child->localName) {
		[...]
		case 'RetrievalMethod':
		$type = $child->getAttribute('Type');
		if ($type !== 'http://www.w3.org/2001/04/xmlenc#EncryptedKey') {
			/* Unsupported key type. */
			break;
		}
		$uri = $child->getAttribute('URI');
		if ($uri[0] !== '#') {
			/* URI not a reference - unsupported. */
			break;
		}
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
$id = substr($uri, 1);

$query = "//xmlsecenc:EncryptedKey[@Id='$id']";
$keyElement = $xpath->query($query)->item(0);
if (!$keyElement) {
       throw new Exception("Unable to locate
              EncryptedKey with @Id='$id'.");
}

return XMLSecurityKey::fromEncryptedKeyElement($keyElement);
case 'EncryptedKey':
return XMLSecurityKey::fromEncryptedKeyElement($child);
```

It has been unveiled that the *staticLocateKeyInfo* function calls *fromEncryptedKeyElement* in some cases. Please pay attention to the fact that the element passed to *fromEncryptedKeyElement* is not necessarily a *child* element of the *$node* element. Under this premise, consider the following implementation of the *fromEncryptedKeyElement*.

Affected File:
xmlseclibs/src/XMLSecurityKey.php

Affected Source:

```
public static function fromEncryptedKeyElement(DOMElement $element)
{

       $objenc = new XMLSecEnc();
       $objenc->setNode($element);
       if (! $objKey = $objenc->locateKey()) {
       throw new Exception("Unable to locate algorithm for this Encrypted Key");
       }
       $objKey->isEncrypted = true;
       $objKey->encryptedCtx = $objenc;
       XMLSecEnc::staticLocateKeyInfo($objKey, $element);
       return $objKey;
}
```

This function makes use of the *staticLocateKeyInfo* in order to locate the desired element in the XML document. An attacker could therefore build a SAML document, which triggers an endless loop for these functions. This results in a DoS condition. One simple way of avoiding this issue is to limit the recursion depth of both functions by passing a counter parameter which increases upon each call. The functions could check the value of this parameter and base their next actions on this. Specifically, if the parameter value exceeds a hard-coded limit, processing of the SAML document could be aborted.

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

SSP-01-004 SAML2: DoS in the Timestamp Function *(Info)*

A regular expression within a *timestamp conversion* function may foster a Denial of Service since it permits the insertion of an arbitrarily large *timestamp*.

Affected File:
saml2/src/SAML2/Utils.php

Affected Source:

```
public static function xsDateTimeToTimestamp($time) {
	$matches = array();
	// We use a very strict regex to parse the timestamp.
	$regex = '/^(\\d\\d\\d\\d)-(\\d\\d)-(\\d\\d)T(\\d\\d):(\\d\\d):
		(\\d\\d)(?:\\.\\d+)?Z$/D';
	if (preg_match($regex, $time, $matches) == 0) {
		throw new \Exception(
		'Invalid SAML2 timestamp passed to xsDateTimeToTimestamp: ' .
$time
		);
	}
[...]
}
```

It is recommended to ensure that the regular expression pointed out above is modified to rule out inputs of arbitrary length. This can be accomplished by, for instance, using a quantifier for length restrictions.

SSP-01-005 SimpleSAMLphp: checkURLAllowed can be bypassed *(Medium)*

The *checkURLAllowed* utility function uses a loose *RegExp* to parse the input URL. A specially crafted URL can confuse the *RegExp* and have it extract the wrong URL parts. Since this function is used to check if a URL is whitelisted, an attacker can abuse the flaw and perform a malicious redirection.

Affected File:
lib/SimpleSAML/Utils/HTTP.php

Affected Source:

```
if (is_array($trustedSites)) {
    assert(is_array($trustedSites));
    preg_match('@^http(s?)://([^/:]+)((?::\d+)?)@i', $url, $matches);
```

PoC:
https://ssp-demo-idp.paas2.uninett.no/simplesaml/module.php/core/login-admin.php?ReturnTo=http%3A%2F%2Fssp-demo-idp.paas2.uninett.no:@example.com

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

The *ReturnTo* parameter contains a URL that includes a whitelisted URL in the *authority* part of a URL, yet the actual host is *example.com*. A proper URL parsing function should be used instead of *RegExp* which is commonly known as prone to bypasses.

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

SSP-01-006 XMLSEC: Using == For Hash Comparison *(Low)*

The analyzed code makes use of the `==` operator for comparing hash values. An excerpt containing affected code can be consulted next.

Affected File:
xmlseclibs/src/XMLSecurityDSig.php

Affected Source:

```
public function validateDigest($refNode, $data)
{
	$xpath = new DOMXPath($refNode->ownerDocument);
	$xpath->registerNamespace('secdsig', self::XMLDSIGNS);
	$query = 'string(./secdsig:DigestMethod/@Algorithm)';
	$digestAlgorithm = $xpath->evaluate($query, $refNode);
	$digValue = $this->calculateDigest($digestAlgorithm, $data, false);
	$query = 'string(./secdsig:DigestValue)';
	$digestValue = $xpath->evaluate($query, $refNode);
	return ($digValue == base64_decode($digestValue));
}
```

This handling is generally not recommended for two separate reasons. First, the behavior means that a door can be opened for timing attacks. Even though this is not the case in this scenario, it is generally a potentially dangerous implication. Secondly, the `==` operator performs *type* conversions before comparing both values. If both values can be parsed as integers, then they both get converted to integer values before the comparison takes place. Hence, the strings "*1e3*" and "*1000*" would be considered equal (the first string makes us of a scientific number notation).

For hash values, this means that if both values start with "*0e*" and continue with exclusively numerical characters, these iterations will be considered equal again, even if they are in fact different. The likelihood that a random SHA-1 hash value takes the desired form is *(1/256)^2 * (10/256)^18*, which roughly comes to a value of *2^-99*. In

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

other words, no critical impact arises from this handling if SHA-1 is used. It is strongly recommended to make use of the === operator which avoids these kinds of unsafe *type* conversions.

SSP-01-007 XMLSEC: Dangerous Use of file_get_contents *(Low)*

A code excerpt from the analyzed source code is supplied below to demonstrate a case of a dangerous use pertaining to *file_get_* contents.

Affected File:
xmlseclibs/src/XMLSecurityDSig.php

Affected Source:

```
if ($uri = $refNode->getAttribute("URI")) {
       $arUrl = parse_url($uri);
       if (empty($arUrl['path'])) {
       if ($identifier = $arUrl['fragment']) {
              [...]
        } else {
              $dataObject = file_get_contents($arUrl);
        }
} else {
[...]
```

The above code uses *file_get_contents* to load file contents from a resource specified in the provided XML document. This can generally lead to Denial of Service conditions, as it can allow reading local files like */dev/random*. Furthermore, it could possibly result in loading remote files from an attack-provided URI. In this case, the *file_get_contents($arUrl)* operation will never succeed because *$arUrl* is not a string but a parsed URI object.

The issue should ideally be addressed by removing the *file_get_contents* call.

SSP-01-008 SimpleSAMLphp: Use of UTF8 for in SQLAuth *(Info)*

The SQLAuth module uses *utf8* as the connection charset for MySQL. In spite of what its name suggests, the module only supports a subset of code points for Unicode. This could result in vulnerabilities[1] that abuse the fact that MySQL silently truncates invalid values.

Affected File:
modules/sqlauth/lib/Auth/Source/SQL.php

[1] https://mathiasbynens.be/notes/mysql-utf8mb4

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Affected Source:

```
/* Driver specific initialization. */
switch ($driver) {
case 'mysql':
       /* Use UTF-8. */
       $db->exec("SET NAMES 'utf8'");
       break;
```

Two mitigating factors are that the issue mostly affects *INSERT* and *UPDATE* statements and that the module makes use of the *SELECT* statement. It is nevertheless recommended to use "*utf8mb4"*, as this solution fully supports Unicode as the connection charset. A revised approach will help avoid potential issues.

SSP-01-009 SimpleSAMLphp: Potential XSS due to inaccurate URL filtering *(Low)*

It was found that many template files make use of the wrong filter function for URLs. Specifically, the pattern presented next is heavily used.

Affected Code:

```
<a href="<?php echo htmlspecialchars($this->data['urlAgree']); ?>">
```

Although *htmlspecialchars* prevents breaking out of the attribute, it does not eliminate a potential for injecting XSS payloads that use JavaScript pseudo-protocol. This stems from the fact that the function only converts the input string to HTML entities. By injecting something like *javascript:alert(document.domain)* and having users click on the link, successful XSS can be accomplished.

A proper URL filter should be used and it verify whether the scheme is in fact HTTP/HTTPS.

SSP-01-010 SimpleSAMLphp: Potential XSS due to missing escaping flags *(Low)*

It was discovered that *htmlspecialchars* is missing the *ENT_QUOTES* flag as a parameter on one occasion. If no *ENT_QUOTES* is specified, the function will still replace double quotes, however single quotes are left untouched. In the code below, this becomes an issue because an attacker can break out of the *href* attribute and achieve JavaScript execution by specifying a malicious *on-handler* attribute, e.g. *onclick*.

Affected File:
simplesamlphp-master/modules/consent/templates/consentform.php

Affected Code:

```
if ($this->data['sppp'] !== false) {
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
    echo "<p>" . htmlspecialchars($this-
>t('{consent:consent:consent_privacypolicy}')) . " ";
    echo "<a target='_blank' href='" . htmlspecialchars($this->data['sppp']) .
"'>" . $dstName . "</a>";
    echo "</p>";
}
```

During the test it was not possible to influence the *sppp* value in order to exploit this behavior. However, it is considered good practice to use *ENT_QUOTES* wherever possible and it is recommended to enforce this throughout the codebase.

Conclusions

The results of this November 2017 penetration tests and audits of the SimpleSAMLphp are fairly good. Seven testers from Cure53 and Secfault Security were quite positive about the overall ten findings on the targeted product, especially since the project entails an application based on the commonly risky PHP framework.

Among the findings, five were classified as security vulnerabilities and the remaining five were deemed to be general weaknesses. Only one discovery was rated to be of the utmost “Critical” severity given its great severity and impact. This problem was specifically caused by a flaw in the *xmlsec* library, which was proven to allow signature bypasses under certain circumstances. Due to the heavy use of *xmlsec* in other libraries, this finding was deemed to be a pivotally important issue and flagged for an immediate fix. Other findings like the various XPATH injections or weak comparisons in *xmlsec* also posed serious risks but had a lower significance for this audit, thus receiving “Medium” rankings.

The majority of the testing efforts were dedicated to the core scope within SimpleSAMLphp app, especially the web interface and all default modules connected to the main item. These components were all audited and the code made a strong impression. Even the easy to overlook PHP features like strong type comparisons seem to be mitigated and used properly by the SimpleSAMLphp project maintainers. Similarly strong stature with no issues characterized the modules that were not explicitly enabled by default but otherwise signaled extensive usage - for instance *Oauth/2* and *logpeek.* Conversely, one core functionality checking whether a user of the SimpleSAMLphp supplied URLs was found prone to issues. Specifically, it was bypassed and led to an array of potential issues in the project.

Covered by the audit were also the libraries like *saml2* but, once again, no issues beyond minor *RegExp* DoS were spotted in this realm. While the cryptographic components of this software library were generally implemented in a sound and prudent

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

fashion, one major cryptographic vulnerability was an exception to this trend. In SSP-01-001, one can observe that cryptographic signature checks can be bypassed by an active attacker. The main issue at hand was that *asymmetric* cryptographic signature algorithms were being confused to be in the same category as *symmetric* authentication code algorithms. Although the functionality of these two types of cryptographic primitives does frequently overlap, the *symmetric* nature of HMACs means that a fundamentally different API is needed. When the *asymmetric* API for signatures is overloaded to support HMAC, a simple change of the HMAC parameters sent over the wire signifies an attacker having complete control over which messages can be validated. Aside from this major design error, the cryptographic primitives were found to be reasonably in line with what can be expected from a standard SAML implementation.

It is believed that this Cure53 test of the SimpleSAMLphp sponsored by Mozilla can be a stepping stone for the project to become even more secure. It should be noted that the aforementioned issues must be addressed as soon as possible, though some of the items reported here were of a rather “Low” impact, mainly because the code was just weak in terms of incorrectly used output-validation. This is not to say that these instances can be ignored but rather that preventing vulnerabilities is a work in progress and it is highly recommended to fix the highlighted code paths as part of a newly revised and additional hardening approaches. All in all, Cure53 believes SimpleSAMLphp to be on the right track from a security standpoint.

Cure53 would like to thank Jaime Perez Crespo from the SimpleSAMLphp team for his excellent project coordination, support and assistance, both before and during this assignment.