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

Pentest-Report php-saml-sp Libraries 10.-11.2020

Cure53, Dr.-Ing. M. Heiderich, BSc. T.-C. “Filedescriptor” Hong, Dipl.-Ing. M. Vervier, MSc. L. Gommans

Index

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

Introduction

"There are various options for integrating SAML in your PHP application. However, most are either (very) complicated, include too many (useless) features, have hard requirements on Apache and are not easy to package for server operating systems like CentOS/Fedora and/or Debian. We only need SAML SP support, so there is no need to include any IdP features, or other (obsolete) authentication protocols. In addition, we only implement what is actually used "in the field" and that which is secure. So you won't find SHA1 support or insecure encryption."

From https://git.tuxed.net/fkooman/php-saml-sp/about/

This report details the scope, findings and mitigatory advice concerning a penetration test and source code audit against the *php-saml-sp* Secure SAML Service Provider, a software written in PHP that is available as Open Source Software (OSS).

The work was requested by the Danish e-Infrastructure Cooperation (DeIC) and executed by the Cure53 team in late October and early November 2020, namely during CW44 and CW45. The aforementioned team consisted of four senior testers, who spent a total of twelve days on the scope to reach an optimal level of reporting coverage.

The work was divided into three distinct yet related work packages (WPs) for efficient structuring. These were:

- WP1: General tests & audits against *php-saml-sp* & *php-secookie*
- WP2: SAML & XML-focused tests & source code audits of *php-saml-sp*
- WP3: Penetration tests & assessments of the deployed *php-saml-sp* software

The chosen methodology for this project was white-box. The involved Cure53 members were granted access to all relevant sources (which are available as OSS) as well as a testing environment furnished by the maintainers.

Communications during the tests and audits were facilitated via a dedicated Signal group that was created by the maintainers and within which the testers were invited to participate. Communications were productive and supportive, allowing for an optimum environment in which any and all issues could be reported and discussed simultaneously with the ongoing testing. In fact, such was the team's efficiency during this period that a handful of issues were correctly fixed during the testing phase - these are marked in this report with a note.

The exemplary preparations and effective channels of communication contributed toward ensuring the Cure53 team made above-par headway, thereby enabling outstanding coverage levels. This is reflected by the findings detected during this test. Specifically, the tests and audits unveiled a total of seven anomalies: three of which were classified to be security vulnerabilities, and four to be general weaknesses of lower exploitation potential.

The overall impression gained is sufficiently positive, as none of the issues identified were categorized with a higher severity level than *Medium*. Having said this, one could deem the Remote Code Execution in DEI-01-003 a 'close call' and a potential concern, but thankfully this issue proved to be unexploitable in actuality.

The report will now shed some light on the scope and test setup, before listing all findings by group and then in chronological order. Each finding will be accompanied by a technical description and a PoC where possible, plus mitigation or fix advice if necessary. Subsequently, the report will finalize with a conclusory summation in which the Cure53 team will elaborate on the general impressions gained over the course of this test and audit, before relaying broader and tailored high-level hardening advice.

***Note**: This report was updated in late November 2020 after Cure53 was able to successfully perform a fix verification process in collaboration with the developers.*

Each issue ticket has been updated with a note to clarify on the status of the respective fix or mitigation. Fixes have been verified based on diffs and detailed descriptions.

Scope

- **White-Box Tests & Audits against *php-saml-sp* Secure SAML Service Provider**
  - **WP1:** General tests & audits against *php-saml-sp* & *php-secookie*
    - https://git.tuxed.net/fkooman/php-saml-sp
    - https://git.tuxed.net/fkooman/php-secookie
  - **WP2**: SAML & XML-focused tests & source code audits of *php-saml-sp*
    - See above
  - **WP3**: Penetration tests & assessments of the deployed *php-saml-sp* software
    - **Test system**
      - https://debianx.tuxed.net/
    - **SP landing page**
      - https://debianx.tuxed.net/php-saml-sp/
  - **Sources were shared with Cure53**

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues detected throughout the testing period. Please note that findings are listed in 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. *DEC-01-001*) for the purpose of facilitating any future follow-up correspondence.

DEC-01-001 WP1: *ReturnTo* validation bypass via URL parser problem (*Medium*)

***Note**: This issue has been addressed successfully by the development team. The problem as described no longer exists. The fix was verified by Cure53.*

A bypass of the open redirect protection on the *login* endpoint was discovered during the test. The URL parameter named *"ReturnTo"* can be provided to the login HTTP *GET* request to set the URL that a user is redirected to after successful login. The value is validated by the function *verifyReturnToOrigin()* that is invoking the *parse_url()* function of PHP. The parsed URL is utilized to identify both the URL scheme and the host of the provided return URL value. These are subsequently validated against a given, expected *origin* value.

Owing to a multitude of discrepancies between parsers of common web browsers (Chromium, Edge, Firefox, Safari) and the PHP URL parser implementation, URLs can be provided that will be parsed differently by a browser from that of the PHP URL. For example, the URL *"https://a\@example.com"* will give host *"a"* in common browsers, while *parse_url()* will normalize the URL to include the host *"example.com"*.

Affected File:
src/Web/Service.php

Affected Code:
```
/**
 * @param string $expectedOrigin
 * @param string $returnTo
 *
 * @return string
 */
private static function verifyReturnToOrigin($expectedOrigin, $returnTo)
{
    // Origin is scheme://host[:port]
    if (false === \filter_var($returnTo, FILTER_VALIDATE_URL)) {
        throw new HttpException(400, 'invalid "ReturnTo" provided');
    }
```

```
        if (false === $parsedUrl = \parse_url($returnTo)) {
            throw new HttpException(400, 'invalid "ReturnTo" provided');
        }
         // the filter_var FILTER_VALIDATE_URL make sure scheme and host are
        // there, but just to make absolutely sure...
        if (!\array_key_exists('scheme', $parsedUrl) || !\array_key_exists('host',
           $parsedUrl)) {
            throw new HttpException(400, 'invalid "ReturnTo" provided');
        }
        $urlConstruction = $parsedUrl['scheme'].'://'.$parsedUrl['host'];
        if (\array_key_exists('port', $parsedUrl)) {
            $urlConstruction .= ':'.(string) $parsedUrl['port'];
        }
        if ($expectedOrigin !== $urlConstruction) {
            throw new HttpException(400, 'invalid "ReturnTo" provided');
        }
        return $returnTo;
    }
```

An attacker who can bypass the URL validation may provide malicious URLs via a crafted *GET* parameter. This can allow adversaries to manipulate users via URLs that look harmless but contain an encoded parameter, as shown above. It is recommended to end the reliance on the original value provided via the *GET* parameter *"ReturnTo"*. Instead, the URL could be normalized before validation allowing for the implementation of the normalized value only. This way, the potential for mismatching between the various parser implementations is greatly reduced, since the normalized value would not contain any special and escaped characters, nor any other ambiguous elements.

DEC-01-002 WP1: XSS in example code via unencoded SAML attributes *(Medium)*

***Note**: The issue was fixed during the audit and the fix was verified by Cure53.*

Testing confirmed that the example code provided for users fails to encode SAML attributes returned in an SAML assertion. An attacker could potentially craft SAML assertions which contain XSS payloads with HTML characters.

Affected File:
src/example/index.php

Affected Code:
```
foreach ($samlAssertion->getAttributes() as $k => $v) {
    echo $k.': '.\implode(',', $v).'<br>';
}
```

It is recommended to encode SAML attributes and all user-input in general.

DEC-01-007 WP1: CSRF in *logout due* to missing CSRF protection *(Low)*

***Note**: This issue has been addressed successfully by the development team. The problem as described no longer exists. The fix was verified by Cure53.*

Neither the php-saml-sp project nor the demo application employ anti-Cross-Site Request Forgery (CSRF) tokens. For SAML, this is unnecessary on the whole because the requests are not replayable - however, the demo application has alternative actions which modify states, for instance *logout*.

The *Referer* header is reviewed on *POST* requests, yet the *logout* URL is accessible through a *GET* request. Since the *logout* is the most effective action an attacker can take via the means of CSRF, the impact is limited to solely disallowing a user to access the application while an attacker's web page is open. Nevertheless, CSRF in general lets attackers perform actions on the user's behalf and manipulate server state without having access to the *authentication* cookie.

PoC:
The following *image t*ag can be placed in an HTML file. In the eventuality that this file is opened by a logged-in user, the user will be logged out.

```
<img src="https://debianx.tuxed.net/php-saml-sp/logout?ReturnTo=https%3A%2F
%2Fdebianx.tuxed.net%2Fvpn-user-portal%2Faccount">
```

Several mitigations are possible here. A regular application will seek to employ anti-CSRF tokens or consistently check the *Referer* header, but a simple solution for the *php-saml-sp* library would be to disallow *GET* and other methods to be used for the *logout* request, since the *Referer* is already being checked for *POST* requests.

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

Miscellaneous Issues

This section covers any noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious intent in the future. The majority 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.

DEC-01-003 SP: Deserialization could lead to Remote Code Execution *(Low)*

The *php-saml-sp* project was found to use PHP's built-in serialization to store data in the user's session. In this way, no method was found through which the user could control the session data. However, the function *handleResponse* in *SP.php* retrieves a value from the session in which the user is able to control the relevant part of the key, then deserializes the value. If the user was able to control the data in *any* prefixed session key - for example, if the software would store the user's name in the session - this could result in remote code execution.

Affected File:
src/SP.php

Affected Code:
```
public function handleResponse($samlResponse, $relayState)
{
    if (null === $sessionValue = $this->session->take(self::SESSION_KEY_PREFIX.
$relayState)) {
        throw new SpException('"RelayState" not found in session data');
    }
    $authnRequestState = \unserialize($sessionValue);
```

The PHP *unserialize* documentation warns not to use this function toward untrusted user input. While this is not currently the case, changes in seemingly-unrelated sections of the code could enable this to occur. PHP considers JSON to be a safe data-interchange format. Using the *json_encode* and *-decode* functions rather than object serialization will prevent this becoming exploitable. To maintain the existing type check, a *type* field could be used in the JSON data.

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

DEC-01-004 SP: User can inject XML which is subsequently signed *(Low)*

***Note**: This issue has been addressed successfully by the development team. The problem as described no longer exists. The fix was verified by Cure53.*

A user can inject XML into a template, which is then signed by the application and returned to the user. No purpose was found for this signed data, as the IDP does not verify it before parsing and, if they did, it would only facilitate attack toward the IDP by enabling the supply of altered XML data.

The templating engine in *src/web/Tpl.php* commences output buffering, *includes* (executes) the template which may use global variables, then retrieves and cleans the output buffer. The template *AuthnRequest.php* echoes user-supplied data without escaping it. The function *prepareRequestUrl* signs the resulting value and it is returned to the user in the form of an HTTP redirect.

Affected File:
AuthnRequest.php

Affected Code:
```
  <samlp:RequestedAuthnContext Comparison="exact">
<?php foreach ($AuthnContextClassRef as $v): ?>
       <saml:AuthnContextClassRef><?=$v; ?></saml:AuthnContextClassRef>
<?php endforeach; ?>
  </samlp:RequestedAuthnContext>
<?php endif; ?>
<?php if (0 !== \count($ScopingIdpList)): ?>
  <samlp:Scoping>
       <samlp:IDPList>
<?php foreach ($ScopingIdpList as $ScopingIdp): ?>
       <samlp:IDPEntry ProviderID="<?=$ScopingIdp; ?>"/>
<?php endforeach; ?>
       </samlp:IDPList>
```

PoC:
https://debianx.tuxed.net/php-saml-sp/login?ScopingIdpList=">&ReturnTo=https://debianx.tuxed.net/

It is recommended to escape values for the target syntax - in this case XML - each and every time, or alternatively to keep code and data separate by inserting nodes rather than concatenating strings. Furthermore, running XML templates through the PHP interpreter might lead to alternative bugs - such as the instance by which short open tags are enabled, and an XML processing instruction or XML declaration is included in the template.

DEC-01-005 SP: Potential signature bypass via empty string *C14N()* failure *(Low)*

***Note**: This issue has been addressed successfully by the development team. The problem as described no longer exists. The fix was verified by Cure53.*

Before signature checking, the XML data is canonicalized using the *DOMNode::C14N* method. According to the documentation of *C14N*[1], the function can return the Boolean value *false* in case of an error.

Affected File:
src/Crypto.php line 69, line 232, and line 236

Affected Code:

```
$signedInfoElement =
$xmlDocument->requireOneDomElement('self::node()/ds:Signature/ds:SignedInfo');
$canonicalSignedInfo = $signedInfoElement->C14N(true, false);
$signatureElement =
$xmlDocument->requireOneDomElement('self::node()/ds:Signature');
$rootElement = $xmlDocument->requireOneDomElement('self::node()');
$rootElement->removeChild($signatureElement);
$rootElementDigest = Base64::encode(
     \hash(
         self::SIGN_HASH_ALGO,
         self::canonicalizeElement($rootElement),
         true
     )
 );

 // compare the digest from the XML with the actual digest
 if (!\hash_equals($rootElementDigest, $digestValue)) {
     throw new CryptoException('unexpected digest');
 }

self::verify($canonicalSignedInfo, Base64::decode($signatureValue), 
$publicKeys);
}
```

The value *false* will be converted into the empty string when used in a string context. This means that an attacker, knowing a valid signature for a signature info block containing the SHA256 hash of the empty string, could supply any XML fragment that

[1] https://www.php.net/manual/de/domnode.c14n.php

will make *C14N* return *false* to the signature check and pass it successfully.

The finding's impact categorization is minimized here due to two reasons: the requirement to be privy to a valid signature for the empty string, and the requirement to supply an XML fragment that fails canonicalization via *C14N.*
It is still recommended to handle any failures of function *C14N,* to ensure that only canonicalized XML content is processed after the initial signature verification step, and that error-return values are handled explicitly in case of failed canonicalization.

DEC-01-006 SP: Non-canonicalized XML data used after signature check *(Low)*

***Note**: This issue has been addressed successfully by the development team. The problem as described no longer exists. The fix was verified by Cure53.*

An attacker can create an XML document that, when canonicalized, matches the signed hash and passes the signature checks. If the document contains data that is processed during canonicalization, this data could enable attacks in subsequent processing steps. The document could potentially be processed later by another parser that reads the metadata file written in *MetadataSource.php* line 228 to a disk via file.

Affected File:
MetadataSource.php line 228 general parsing code strategy

Affected Code:
```
// verify the metadata schema and signature
$metadataDocument = XmlDocument::fromMetadata($responseBody, true);
Crypto::verifyXml($metadataDocument, $publicKeyList);

// write metadata to disk using temporary file trick for "atomic" file
// update so the metadata file can't get corrupted
if (false === $tmpFile = \tempnam(\sys_get_temp_dir(), 'php-saml-sp')) {
    throw new RuntimeException('unable to generate a temporary file');
}
if (false === @\file_put_contents($tmpFile, $responseBody)) {
    throw new RuntimeException(\sprintf('unable to write "%s"', $tmpFile));
}
if (false === @\rename($tmpFile, $metadataFile)) {
    throw new RuntimeException(\sprintf('unable to move "%s" to "%s"',
      $tmpFile, $metadataFile));
}

if (null !== $lastModified = $httpClientResponse->getHeader('Last-Modified'))
{
     // use Last-Modified header to set the metadata file's modified
     // time, if available from server to be used on future requests as
```

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

```
        // the "If-Modified-Since" header value
        $lastModifiedDateTime = new DateTime($lastModified);
        if (false === $lastModifiedTimestamp =
         $lastModifiedDateTime->getTimestamp()) {
            $lastModifiedTimestamp = 0; // 1970-01-01
        }
        \touch($metadataFile, $lastModifiedTimestamp);
}
$this->writeRefreshAt($metadataFile, $metadataDocument);
$this->logger->notice(\sprintf('[%s] metadata updated', $metadataUrl))
```

During canonicalization, an XML fragment is transformed via defined methods to a normalized form that should be comparable. This means that whitespace and other data can be removed from the document. Even though the canonicalization is specified and defined by XML standards, subtle differences in various parser implementations might occur that could lead to differing results.

By applying the signature checks to the canonicalized form of the XML data using the original data in subsequent processing steps, attackers may be able to insert additional data into the signed XML document, thereby leveraging the potential for attack. Such an attack could not be implemented in practice within the timeframe given; nevertheless, such instances have been confirmed in previous reports[2]. It is recommended to use the canonicalized XML data only for further processing after the cryptographic signature has been verified.

[2] https://cheatsheetseries.owasp.org/cheatsheets/XML_Security_Cheat_Sheet.html

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

Conclusions

The impressions gained during this report - which details and extrapolates on all findings identified during the CW44 and CW55 testing against the php-saml-sp Secure SAML Service Provider by the Cure53 team - will now be discussed at length. To summarize, it can be considered that the target software functions have left a positive impression with no severity ratings higher than *Medium* recorded.

For ease of reading, the conclusory assessments will now be split into two distinct sections, first in relation to impressions regarding application security posture, then moving to code audit and code quality.

In the first area, standard web application security was performed during the testing phase. The confirmation was made that optimal HTTP security headers are set and no CSRF/XSS issues exist. A reflected XSS was found in the example code via SAML attributes (as reported in DEC-01-002).

An extensive focus was placed toward XML handling and signature wrapping attacks in particular. All known signature wrapping attacks found in other SAML libraries were tested against *php-saml-sp*. Due to the fact that the software supports signing the whole assertion solely, the potential for signature wrapping attacks is minimal. In addition, general SAML validation was scrutinized. Actions toward ensuring an effective reading here included ensuring values such as *Issuer, AudienceRestriction, InResponseTo, NotOnOrAfter* and so on, are correctly deployed and assessed.

As noted, the second area here concerns the php-saml-sp code. Specifically, a source code audit was performed simultaneously with standard web application security checks. Some dependencies of interest, specifically *paragonie/random_compat* and *paragonie/constant_time_encodin*g, were also checked for common issues. Certain complex PHP functions that accept external input, such as *gzdeflate*, were tested to a limited extent to determine whether weaknesses have the potential to materialize in the future.

The majority of code snippets demonstrate a strong level of security awareness, and the absence of severe issues attests to the overall quality of the project. One instance worth highlighting was the handling of template files: the templates are evaluated using a construction based on output buffering, and variables are employed across files. This obscures the fact that variables are being injected through evaluated code, and likely explains the existence of the issue detailed by DEC-01-004.

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

Also pertinent to mention here: the verification of XML signatures is performed on the canonicalized form of the XML data, whilst the actual processing and handling of the XML data is actioned on the original XML data. This can lead to partial signature bypasses such as those described in findings DEC-01-005 and DEC-01-006.

Another particular section resulting in a positive reading was prevalent in the used *random_compat* library. One often finds that random libraries regress to insecure random-number generation if the absence of a cryptographically-secure generator occurs. In this instance, such a failure would result in an exception rather than silently continuing.

In summation, the impressions gained from this autumn 2020 project on the whole were relatively positive. No issues beyond a *Medium* severity rating were detected; one sole vulnerability was close to being categorized as *Critical* but ultimately resulted in being classified as an unexploitable miscellaneous issue instead. Furthermore, in Cure53's view, the in-house team appears to have a firm grasp on current, optimum development practices. Once all vulnerabilities have been mitigated, the application should enjoy the fruits of a strong security infrastructure, as alluded to by the positive results detailed within this report.

Cure53 would like to thank François Kooman, Tangui Coulouarn, Rogier Spoor and Mads Freek Petersen from the Danish e-Infrastructure Cooperation team for their excellent project coordination, support and assistance, both before and during this assignment.