OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: July 26, 2015

N. Sakimura, Ed.
Nomura Research Institute
J. Bradley
Ping Identity
N. Agarwal
Google

January 22, 2015

Proof Key for Code Exchange by OAuth Public Clients
draft-ietf-oauth-spop-06

Abstract

OAuth 2.0 public clients utilizing the Authorization Code Grant are susceptible to the authorization code interception attack. This specification describes the attack as well as a technique to mitigate against the threat.

Status of This Memo

This Internet-Draft will expire on July 26, 2015.

Copyright Notice

Table of Contents

1. Introduction

OAuth 2.0 [RFC6749] public clients are susceptible to the authorization "code" interception attack.

The attacker thereby intercepts the authorization code returned from the authorization endpoint within communication path not protected by

TLS, such as inter-app communication within the operating system of the client.

Once the attacker has gained access to the authorization code it can use it to obtain the access token.

Figure 1 shows the attack graphically. In step (1) the native app running on the end device, such as a smart phone, issues an authorization request via the browser/operating system, which then gets forwarded to the OAuth 2.0 authorization server in step (2). The authorization server returns the authorization code in step (3). The malicious app is able to observe the authorization code in step (4) since it is registered to the custom URI scheme used by the legitimate app. This allows the attacker to reguest and obtain an access token in step (5) and step (6), respectively.

```
    +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
    | End Device (e.g., Smart Phone) |
    |                                |
    | +-------------+   +----------+ | (6) Access Token  +----------+
    | |Legitimate   |   | Malicious|<--------------------|          |
    | |OAuth 2.0 App|   | App      |-------------------->|          |
    | +-------------+   +----------+ | (5) Authorization |          |
    |        |    ^          ^       |        Grant      |          |
    |        |     \         |       |                   |          |
    |        |      \   (4)  |       |                   |          |
    |    (1) |       \  Authz|       |                   |          |
    |   Authz|        \ Code |       |                   |  Authz   |
    | Request|         \     |       |                   |  Server  |
    |        |          \    |       |                   |          |
    |        |           \   |       |                   |          |
    |        v            \  |       |                   |          |
    | +----------------------------+ |                   |          |
    | |                            | | (3) Authz Code    |          |
    | |     Operating System/      |<--------------------|          |
    | |         Browser            |-------------------->|          |
```

```
  | |                            | | (2) Authz Request |           |
  | +----------------------------+ |                   +-----------+
  +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
```

Figure 1: Authorization Code Interception Attack.

A number of pre-conditions need to hold in order for this attack to work:

1) The attacker manages to register a malicious application on the client device and registers a custom URI scheme that is also used by another application.

The operating systems must allow a custom URI schemes to be registered by multiple applications.

2) The OAuth 2.0 authorization code grant is used.

3) The attacker has access to the client id. All native app client-instances use the same client id. No client secret is used (since public clients cannot keep their secrets confidential.)

4) The attacker (via the installed app) is able to observe responses from the authorization endpoint. As a more sophisticated attack scenario the attacker is also able to observe requests (in addition to responses) to the authorization endpoint. The attacker is, however, not able to act as a man-in-the-middle.

While this is a long list of pre-conditions the described attack has been observed in the wild and has to be considered in OAuth 2.0 deployments. While Section 4.4.1 of [RFC6819] describes mitigation techniques they are, unfortunately, not applicable since they rely on a per-client instance secret or aper client instance redirect URI.

To mitigate this attack, this extension utilizes a dynamically created cryptographically random key called 'code verifier'. The code verifier is created for every authorization request and its transformed value, called 'code challenge', is sent to the authorization server to obtain the authorization code. The authorization "code" obtained is then sent to the token endpoint with the 'code verifier' and the server compares it with the previously received request code so that it can perform the proof of possession of the 'code verifier' by the client. This works as the mitigation since the attacker would not know this one-time key.

1.1. Protocol Flow

```
       +--------+                                  +----------------+
       |        |--(A)-- Authorization Request --->|                |
       |        |        + t(code_verifier), t     |   Resource     |
       |        |                                  |     Owner      |
       |        |<-(B)--- Authorization Grant -----|                |
       |        |                                  +----------------+
       | Client |
       |        |                                  +----------------+
       |        |--(C)--- Access Token Request --->|                |
       |        |          + code_verifier         | Authorization  |
       |        |                                  |     Server     |
       |        |<-(D)------ Access Token ---------|                |
       +--------+                                  +----------------+
```

Figure 2: Abstract Protocol Flow

This specification adds additional parameters to the OAuth 2.0 Authorization and Access Token Requests, shown in abstract form in Figure 1.

A. The client creates and records a secret named the "code_verifier", and derives a transformed version "t(code_verifier)" (referred to as the "code_challenge") which is sent in the OAuth 2.0 Authorization Request, along with the transformation method "t".
B. The resource owner responds as usual, but records "t(code_verifier)" and the transformation method.
C. The client then sends the code to the Access Token Request as usual, but includes the "code_verifier" secret generated at (A).
D. The authorization server transforms "code_verifier" and compares it to "t(code_verifier)" from (B). Access is denied if they are not equal.

An attacker who intercepts the Authorization Grant at (B) is unable to redeem it for an Access Token, as they are not in possession of the "code_verifier" secret.

2. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in Key words for use in RFCs to Indicate Requirement Levels [RFC2119]. If these words are used without being spelled in uppercase then they are to be interpreted with their normal natural language meanings.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC5234].

BASE64URL(OCTETS) denotes the base64url encoding of OCTETS, per Section 3 producing a ASCII [RFC0020] STRING.

BASE64URL-DECODE(STRING) denotes the base64url decoding of STRING, per Section 3, producing a UTF-8 sequence of octets.

SHA256(STRING) denotes a SHA2 256bit hash [RFC6234] of STRING.

UTF8(STRING) denotes the octets of the UTF-8 [RFC3629] representation of STRING.

ASCII(STRING) denotes the octets of the ASCII [RFC0020] representation of STRING.

The concatenation of two values A and B is denoted as A || B.

3. Terminology

In addition to the terms defined in OAuth 2.0 [RFC6749], this specification defines the following terms:

code verifier A cryptographically random string that is used to correlate the authorization request to the token request.

code challenge A challenge derived from the code verifier that is sent in the authorization request, to be verified against later.

Base64url Encoding Base64 encoding using the URL- and filename-safe character set defined in Section 5 of RFC 4648 [RFC4648], with all trailing '=' characters omitted (as permitted by Section 3.2) and without the inclusion of any line breaks, whitespace, or other additional characters. (See Appendix A for notes on implementing base64url encoding without padding.)

4. Protocol

4.1. Client creates a code verifier

The client first creates a code verifier, "code_verifier", for each OAuth 2.0 [RFC6749] Authorization Request, in the following manner:

code_verifier = high entropy cryptographic random ASCII [RFC0020] octet sequence using the url and filename safe Alphabet [A-Z] / [a-z] / [0-9] / "-" / "_" from Sec 5 of RFC 4648 [RFC4648], with length less than 128 characters.

ABNF for "code_verifier" is as follows.

```
code-verifier = 42*128unreserved
unreserved = ALPHA / DIGIT / "-" / "_"
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

NOTE: code verifier SHOULD have enough entropy to make it impractical to guess the value. It is RECOMMENDED that the output of a suitable random number generator be used to create a 32-octet sequence. The Octet sequence is then BASE64URL encoded to produce a 42-octet URL safe string to use as the code verifier.

4.2. Client creates the code challenge

The client then creates a code challenge, "code_challenge", derived from the "code_verifier" by using one of the following transformations on the "code_verifier":

plain "code_challenge" = "code_verifier"

S256 "code_challenge" = BASE64URL(SHA256("code_verifier"))

It is RECOMMENDED to use the S256 transformation when possible.

ABNF for "code_challenge" is as follows.

```
code-challenge = 42*128unreserved
unreserved = ALPHA / DIGIT / "-" / "_"
```

```
ALPHA = %x41-5A / %x61-7A
DIGIT = %x30-39
```

4.3. Client sends the code challenge with the authorization request

The client sends the code challenge as part of the OAuth 2.0 [RFC6749] Authorization Request (Section 4.1.1.) using the following additional parameters:

code_challenge REQUIRED. Code challenge.

code_challenge_method OPTIONAL, defaults to "plain". Code verifier transformation method, "S256" or "plain".

4.4. Server returns the code

When the server issues the "code" in the Authorization Response, it MUST associate the "code_challenge" and "code_challenge_method" values with the "code" so it can be verified later.

Typically, the "code_challenge" and "code_challenge_method" values are stored in encrypted form in the "code" itself, but could alternatively be stored on the server, associated with the code. The server MUST NOT include the "code_challenge" value in client requests in a form that other entities can extract.

The exact method that the server uses to associate the "code_challenge" with the issued "code" is out of scope for this specification.

4.4.1. Error Response

If the server requires PKCE, and the client does not send the "code_challenge" in the request, the authorization endpoint MUST return the authorization error response with "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., code challenge required.

If the server supporting PKCE does not support the requested transform, the authorization endpoint MUST return the authorization error response with "error" value set to "invalid_request". The "error_description" or the response of "error_uri" SHOULD explain the nature of error, e.g., transform algorithm not supported.

If the client is capable of using "S256", it MUST use "S256", as "S256" is MTI on the server. Clients MAY use plain only if they cannot support "S256" for some technical reason and knows that the server supports "plain".

4.5. Client sends the code and the secret to the token endpoint

Upon receipt of the "code", the client sends the Access Token Request to the token endpoint. In addition to the parameters defined in OAuth 2.0 [RFC6749] Access Token Request (Section 4.1.3.), it sends the following parameter:

code_verifier REQUIRED. Code verifier

4.6. Server verifies code_verifier before returning the tokens

Upon receipt of the request at the Access Token endpoint, the server verifies it by calculating the code challenge from received "code_verifier" and comparing it with the previously associated "code_challenge", after first transforming it according to the "code_challenge_method" method specified by the client.

If the "code_challenge_method" from Section 4.2 was "S256", the received "code_verifier" is first hashed with SHA-256 then compared to the base64url decoded "code_challenge". i.e.,

SHA256("code_verifier") == BASE64URL-DECODE("code_challenge").

If the "code_challenge_method" from Section 4.2 was "plain", they are compared directly. i.e.,

"code_challenge" == "code_verifier".

If the values are equal, the Access Token endpoint MUST continue processing as normal (as defined by OAuth 2.0 [RFC6749]). If the values are not equal, an error response indicating "invalid_grant" as described in section 5.2 of OAuth 2.0 [RFC6749] MUST be returned.

5. Compatibility

Server implementations of this specification MAY accept OAuth2.0 Clients that do not implement this extension. If the "code_verifier" is not received from the client in the Authorization Request, servers supporting backwards compatibility SHOULD revert to a normal OAuth 2.0 [RFC6749] protocol.

As the OAuth 2.0 [RFC6749] server responses are unchanged by this specification, client implementations of this specification do not need to know if the server has implemented this specification or not, and SHOULD send the additional parameters as defined in Section 3. to all servers.

6. IANA Considerations

This specification makes a registration request as follows:

6.1. OAuth Parameters Registry

This specification registers the following parameters in the IANA OAuth Parameters registry defined in OAuth 2.0 [RFC6749].

- Parameter name: code_verifier
- Parameter usage location: Access Token Request
- Change controller: IESG
- Specification document(s): this document

- Parameter name: code_challenge
- Parameter usage location: Authorization Request
- Change controller: IESG
- Specification document(s): this document

- Parameter name: code_challenge_method
- Parameter usage location: Authorization Request
- Change controller: IESG
- Specification document(s): this document

6.2. PKCE Code Challenge Method Registry

This specification establishes the PKCE Code Challenge Method registry.

Additional code_challenge_method types for use with the authorization endpoint are registered with a Specification Required ([RFC5226]) after a two-week review period on the oauth-ext-review@ietf.org mailing list, on the advice of one or more Designated Experts. However, to allow for the allocation of values prior to publication,

the Designated Expert(s) may approve registration once they are satisfied that such a specification will be published.

Registration requests must be sent to the oauth-ext-review@ietf.org mailing list for review and comment, with an appropriate subject (e.g., "Request for PKCE code_challenge_method: example").

Within the review period, the Designated Expert(s) will either approve or deny the registration request, communicating this decision to the review list and IANA. Denials should include an explanation and, if applicable, suggestions as to how to make the request successful.

IANA must only accept registry updates from the Designated Expert(s) and should direct all requests for registration to the review mailing list.

6.2.1. Registration Template

Code Challenge Method Parameter Name:
The name requested (e.g., "example"). Because a core goal of this specification is for the resulting representations to be compact, it is RECOMMENDED that the name be short -- not to exceed 8 characters without a compelling reason to do so. This name is case-sensitive. Names may not match other registered names in a case-insensitive manner unless the Designated Expert(s) state that there is a compelling reason to allow an exception in this particular case.

Change Controller:
For Standards Track RFCs, state "IESG". For others, give the name of the responsible party. Other details (e.g., postal address, email address, home page URI) may also be included.

Specification Document(s):
Reference to the document(s) that specify the parameter, preferably including URI(s) that can be used to retrieve copies of the document(s). An indication of the relevant sections may also be included but is not required.

6.2.2. Initial Registry Contents

This specification registers the Code Challenge Method Parameter names defined in Section 4.2 in this registry.

- Code Challenge Method Parameter Name: "plain"
- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[this document]]

- Code Challenge Method Parameter Name: "S256"

- Change Controller: IESG
- Specification Document(s): Section 4.2 of [[this document]]

7. Security Considerations

7.1. Entropy of the code verifier

The security model relies on the fact that the code verifier is not learned or guessed by the attacker. It is vitally important to adhere to this principle. As such, the code verifier has to be created in such a manner that it is cryptographically random and has high entropy that it is not practical for the attacker to guess. It is RECOMMENDED that the output of a suitable random number generator be used to create a 32-octet sequence.

7.2. Protection against eavesdroppers

Clients MUST NOT try down grading the algorithm after trying "S256" method. If the server is PKCE compliant, then "S256" method works. If the server does not support PKCE, it does not generate error. Only the time that the server returns that it does not support "S256" is there is a MITM trying the algorithm downgrade attack.

"S256" method protects against eavesdroppers observing or intercepting the "code_challenge". If the "plain" method is used, there is a chance that it will be observed by the attacker on the device. The use of "S256" protects against it.

If "code_challenge" is to be returned inside authorization "code" to achieve a stateless server, it has to be encrypted in such a manner that only the server can decrypt and extract it.

7.3. Checking the Server support

Before starting the authorization process, the client SHOULD check if the server supports this specification. Confirmation of the server support may be obtained out-of-band or through some other mechanisms such as the discovery document in OpenID Connect Discovery [OpenID.Discovery]. The exact mechanism on how the client obtains this information, or the action it takes as a result is out of scope of this specification.

7.4. Entropy of the code_verifier

The client SHOULD create a code_verifier with a minimum of 256bits of entropy. This can be done by having a suitable random number generator create a 32-octet sequence. The Octet sequence can then be

Base64url encoded to produce a 42-octet URL safe string to use as a code_challenge that has the required entropy.

Salting is not used in the production of the code_verifier, as the code_chalange contains sufficient entropy to prevent brute force attacks. Concatenating a publicly known value to a code_challenge (with 256 bits of entropy) and then hashing it with SHA256 would actually reduce the entropy in the resulting code_verifier making it easier for an attacker to brute force.

While the S256 transformation is like hashing a password there are important differences. Passwords tend to be relatively low entropy words that can be hashed offline and the hash looked up in a dictionary. By concatenating a unique though public value to each password prior to hashing, the dictionary space that an attacker needs to search is greatly expanded.

Modern graphics processors now allow attackers to calculate hashes in real time faster than they could be looked up from a disk. This eliminates the value of the salt in increasing the complexity of a brute force attack for even low entropy passwords.

7.5. OAuth security considerations

All the OAuth security analysis presented in [RFC6819] applies so readers SHOULD carefully follow it.

8. Acknowledgements

The initial draft of this specification was created by the OpenID AB/Connect Working Group of the OpenID Foundation, most notably by the following people:

- Anthony Nadalin, Microsoft
- Axel Nenker, Deutsche Telekom
- Breno de Medeiros, Google
- Brian Campbell, Ping Identity
- Chuck Mortimore, Salesforce
- Dirk Balfanz, Google
- Eduardo Gueiros, Jive Communications
- Hannes Tschonfenig, ARM
- James Manger, Telstra
- John Bradley, Ping Identity
- Justin Richer, MIT Kerberos
- Josh Mandel, Boston Children's Hospital
- Lewis Adam, Motorola Solutions
- Madjid Nakhjiri, Samsung
- Michael B. Jones, Microsoft

- Nat Sakimura, Nomura Research Institute
- Naveen Agarwal, Google
- Paul Madsen, Ping Identity
- Phil Hunt, Oracle
- Prateek Mishra, Oracle
- Ryo Ito, mixi
- Scott Tomlinson
- Sergey Beryozkin
- Takamichi Saito
- Torsten Lodderstedt, Deutsche Telekom
- William Denniss, Google

9. Revision History

-06

- fix date
- replace spop with pkce for registry and other references

- re #29 change name again
- re #27 removed US-ASCII reference
- re #27 updated ABNF for code_verifier
- resolves #24 added security consideration for salting
- resolves #29 Changed title
- updated reference to RFC4634 to RFC6234 re #27
- changed reference for US-ASCII to RFC20 re #27
- resolves #28 added Acknowledgements
- resolves #27 updated ABNF
- resolves #26 updated abstract and added Hannes figure

-05

- Added IANA registry for code_challenge_method + fixed some broken internal references.

-04

- Added error response to authorization response.

-03

- Added an abstract protocol diagram and explanation

-02

- Copy edits

-01

- Specified exactly two supported transformations
- Moved discovery steps to security considerations.
- Incorporated readability comments by Eduardo Gueiros.
- Changed MUST in 3.1 to SHOULD.

-00

- Initial IETF version.

10. References

10.1. Normative References

[RFC0020] Cerf, V., "ASCII format for network interchange", RFC 20, October 1969.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC6234] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, May 2011.

[RFC6749] Hardt, D., "The OAuth 2.0 Authorization Framework", RFC 6749, October 2012.

10.2. Informative References

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", February 2014.

[RFC6819] Lodderstedt, T., McGloin, M., and P. Hunt, "OAuth 2.0 Threat Model and Security Considerations", RFC 6819, January 2013.

Appendix A. Notes on implementing base64url encoding without padding

This appendix describes how to implement base64url encoding and decoding functions without padding based upon standard base64 encoding and decoding functions that do use padding.

To be concrete, example C# code implementing these functions is shown below. Similar code could be used in other languages.

```
  static string base64urlencode(byte [] arg)
  {
    string s = Convert.ToBase64String(arg); // Regular base64 encoder
    s = s.Split('=')[0]; // Remove any trailing '='s
    s = s.Replace('+', '-'); // 62nd char of encoding
    s = s.Replace('/', '_'); // 63rd char of encoding
    return s;
  }

  static byte [] base64urldecode(string arg)
  {
    string s = arg;
    s = s.Replace('-', '+'); // 62nd char of encoding
    s = s.Replace('_', '/'); // 63rd char of encoding
    switch (s.Length % 4) // Pad with trailing '='s
    {
      case 0: break; // No pad chars in this case
      case 2: s += "=="; break; // Two pad chars
      case 3: s += "="; break; // One pad char
      default: throw new System.Exception(
        "Illegal base64url string!");
    }
    return Convert.FromBase64String(s); // Standard base64 decoder
  }
```

As per the example code above, the number of '=' padding characters that needs to be added to the end of a base64url encoded string without padding to turn it into one with padding is a deterministic function of the length of the encoded string. Specifically, if the length mod 4 is 0, no padding is added; if the length mod 4 is 2, two '=' padding characters are added; if the length mod 4 is 3, one '=' padding character is added; if the length mod 4 is 1, the input is malformed.

An example correspondence between unencoded and encoded values follows. The octet sequence below encodes into the string below, which when decoded, reproduces the octet sequence.

3 236 255 224 193

A-z_4ME

Authors' Addresses

Nat Sakimura (editor)
Nomura Research Institute
1-6-5 Marunouchi, Marunouchi Kitaguchi Bldg.
Chiyoda-ku, Tokyo 100-0005
Japan

Phone: +81-3-5533-2111
Email: n-sakimura@nri.co.jp
URI: http://nat.sakimura.org/

John Bradley
Ping Identity
Casilla 177, Sucursal Talagante
Talagante, RM
Chile

Phone: +44 20 8133 3718
Email: ve7jtb@ve7jtb.com
URI: http://www.thread-safe.com/

Naveen Agarwal
Google
1600 Amphitheatre Pkwy
Mountain View, CA 94043
USA

Phone: +1 650-253-0000
Email: naa@google.com
URI: http://google.com/