OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: May 14, 2015

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

November 12, 2014

Symmetric Proof of Possession for the OAuth Authorization Code Grant draft-ietf-oauth-spop-03

Abstract

The OAuth 2.0 public client utilizing Authorization Code Grant (RFC 6749 - 4.1) is susceptible to the code interception attack. This specification describes a mechanism that acts as a control against this threat.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 14, 2015.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction 2

1.1. Protocol Flow 2
2. Notational Conventions 3
3. Terminology 4
4. Protocol 4
4.1. Client creates a code verifier 4
4.2. Client creates the code challenge 5
4.3. Client sends the code challenge with the authorization request 5
4.4. Server returns the code 5
4.5. Client sends the code and the secret to the token endpoint 6
4.6. Server verifies code_verifier before returning the tokens 6
5. Compatibility 6
6. IANA Considerations 6
6.1. OAuth Parameters Registry 7
7. Security Considerations 7
7.1. Entropy of the code verifier 7
7.2. Protection against eavesdroppers 7
7.3. Checking the Server support 7
7.4. OAuth security considerations 8
8. Acknowledgements 8
9. Revision History 8
10. References 9
10.1. Normative References 9
10.2. Informative References 9
Appendix A. Notes on implementing base64url encoding without paddi 10
Authors' Addresses 11

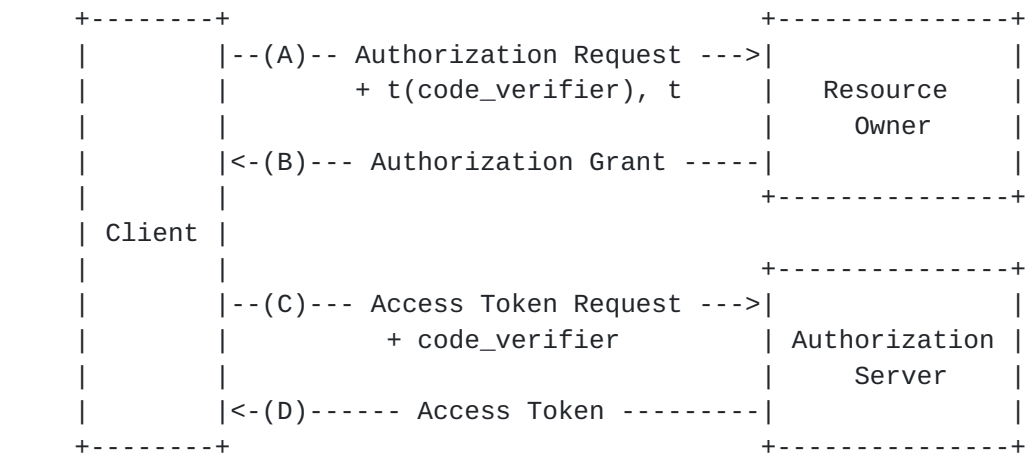
1. Introduction

Public clients in OAuth 2.0 [RFC6749] are susceptible to the authorization "code" interception attack. A malicious client intercepts the authorization code returned from the authorization endpoint and uses it to obtain the access token. This is possible on a public client as there is no client secret associated for it to be sent to the token endpoint. This is especially true on Smartphone applications where the authorization code can be returned through custom URL Schemes where the same scheme can be registered by multiple applications. Under this scenario, the mitigation strategy stated in section 4.4.1 of [RFC6819] does not work as they rely on per-client instance secret or per 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



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 [US-ASCII] 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 [RFC4634] of STRING.

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

ASCII(STRING) denotes the octets of the ASCII [US-ASCII] 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 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 [US-ASCII] 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    = [A-Z] / [a-z] / [0-9] / "-" / "_"
```

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    = [A-Z] / [a-z] / [0-9] / "-" / "_"
```

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.

4.5. 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 3.2 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 3.2 was "none", 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

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

Unless there is a compelling reason, implementations SHOULD use "S256" method to protect against eavesdroppers intercepting the "code_challenge". If the no transformation algorithm, which is the default algorithm, is used, the client SHOULD make sure that the authorization request is adequately protected from an eavesdropper. If "code_challenge" is to be returned inside authorization "code", 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. 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:

- Naveen Agarwal, Google
- Dirk Balfanz, Google
- Sergey Beryozkin
- John Bradley, Ping Identity
- Brian Campbell, Ping Identity
- William Denniss, Google
- Eduardo Gueiros, Jive Communications
- Phil Hunt, Oracle
- Ryo Ito, mixi
- Michael B. Jones, Microsoft
- Torsten Lodderstedt, Deutsche Telekom
- Breno de Medeiros, Google
- Prateek Mishra, Oracle
- Anthony Nadalin, Microsoft
- Axel Nenker, Deutsche Telekom
- Nat Sakimura, Nomura Research Institute

9. Revision History

-03

o Added an abstract protocol diagram and explanation

-02

o Copy edits

-01

o Specified exactly two supported transformations

o Moved discovery steps to security considerations.

o Incorporated readability comments by Eduardo Gueiros.

o Changed MUST in 3.1 to SHOULD.

-00

o Initial IETF version.

10. References

10.1. Normative References

[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.

[RFC4634] Eastlake, D. and T. Hansen, "US Secure Hash Algorithms (SHA and HMAC-SHA)", RFC 4634, July 2006.

[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.

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

[US-ASCII] American National Standards Institute, "Coded Character Set -- 7-bit American Standard Code for Information Interchange", ANSI X3.4, 1986.

10.2. Informative References

[OpenID.Discovery]

Sakimura, N., Bradley, J., Jones, M.B. 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/
```