OAuth Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 22, 2014

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

October 19, 2013

OAuth Symmetric Proof of Posession for Code Extension
draft-sakimura-oauth-tcse-02

Abstract

The OAuth 2.0 public client utilizing authorization code grant is susceptible to the code interception attack. This specification describe a mechanism that acts as a control against this threat.

Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

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 April 22, 2014.

Copyright Notice

Table of Contents

1. Introduction

Public clients in OAuth 2.0 [RFC6749] is suseptible to the "code" interception attack. The "code" interception attack is an attack that a malicious client intercepts the "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 some smartphone platform in which the "code" is returned to a redirect URI with a custom scheme as there can be multiple apps that can register the same scheme.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 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 "code" obtained is then sent to the token endpoint with the code verifier and the server compairs it with the previously received reqeust code so that it can perfom the proof of posession of the code verifier by the client. This works as the mitigation since the attacker would not know the one-time key.

2. Terminology

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

2.1. code verifier

a cryptographically random string with big enough entropy that is used to correlate the authorization request to the token request

2.2. code challenge

either the code verifier itself or some transformation of it that is sent from the client to the server in the authorization request

NOTE 1: The client and the server MAY use mutually agreed pre-negotiated algorithm such as base64url encoding of the left most 128bit of SHA256 hash.

NOTE 2: If no algorithm has been negotiated, it is treated as the code verifier itself.

3. Protocol

3.1. Client checks the server support

Before starting the authorization process, the client MUST make sure that the server supports this specification. It 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 is out of scope of this specification.

The client that wishes to use this specification MUST stop proceeding if the server does not support this extension.

3.2. (optional) Client registers its desired code challenge algorithm

In this specification, the client sends the transformation of the code verifier to the authorization server in the front channel. The default transformation is not doing transformation at all. If the the server supports, the client MAY register its desired transformation algorithm to the server. If the algorithm is registered, the server MUST reject any request that does not conform to the algorithm.

How does this client registers the algorithm is out of scope for this specification.

Also, this specification does not define any transformation other than the default transformation.

3.3. Client creates a code verifier

The client then creates a code verifier, "code_verifier", in the following manner.

code_verifier = high entropy cryptographic random string of length less than 128 bytes

NOTE: code verifier MUST have high enough entropy to make it inpractical to guess the value.

3.4. Client sends the code challenge with the authorization request

Then, the client creates a code challenge, "code_challenge", by applying the pre-negotiated algorithm between the client and the server. The default behavior is no transofrmation, i.e., "code_challenge" == "code_verifier". The authorization server MUST support this 'no transformation' algorithm.

The client sends the code challenge with the following parameter with the OAuth 2.0 [RFC6749] Authorization Request:

code_challenge REQUIRED. code challenge.

3.5. Server returns the code

When the server issues a "code", it MUST associate the "code_challenge" value with the "code" so that it can be used later.

Typically, the "code_challenge" value is stored in encrypted form in the "code", but it could as well be just stored in the server in association with the code. The server MUST NOT include the "code_challenge" value in the form that any entity but itself can extract it.

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

Upon receipt of the "code", the client sends the request to the token endpoint. In addition to the parameters defined in OAuth 2.0 [RFC6749], it sends the following parameter:

code_verifier REQUIRED. code verifier

3.7. Server verifies code_verifier before returning the tokens

Upon receipt of the request at the token endpoint, the server verifies it by calculating the code challenge from "code_verifier" value and comparing it with the previously associated "code_challenge". If they are equal, then the successful response SHOULD be returned. If the values are not equal, an error response indicating "invalid_grant" as described in section 5.2 of OAuth 2.0 [RFC6749] SHOULD be returned.

4. IANA Considerations

This specification makes a registration request as follows:

4.1. 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: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): this document
- Related information: None
- Parameter name: code_challenge
- Parameter usage location: Authorization Request
- Change controller: OpenID Foundation Artifact Binding Working Group - openid-specs-ab@lists.openid.net
- Specification document(s): this document

- Related information: None

5. Security Considerations

The security model relies on the fact that the code verifier is not learned or guessed by the attacker. It is vitally important to adhear 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, and if it is to be returned inside "code", it has to be encrypted in such a manner that only the server can decrypt and extract it.

If the no transformation algorithm, which is the default algorithm, is used, the client MUST make sure that the request channel is adequately protected. On a platform that it is not possible, the client and the server SHOULD utilize a transformation algorithm that makes it reasonably hard to recalculate the code verifier from the code challenge.

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

6. Acknowledgements

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

- Naveen Agarwal, Google
- Dirk Belfanz, Google

- Sergey Beryozkin
- John Bradley, Ping Identity
- Brian Campbell, Ping Identity
- Phill Hunt, Oracle

- Ryo Ito, mixi
- Michael B. Jones, Microsoft
- Torsten Lodderstadt, Deutsche Telekom
- Breno de Madeiros, Google
- Prateek Mishra, Oracle
- Anthony Nadalin, Microsoft
- Axel Nenker, Deutsche Telekom
- Nat Sakimura, Nomura Research Institute

7. Revision History

-02

- Changed title.
- Changed parameter names.
- Changed the default transformation algorithm and added crypto agility.
- More text in the security consideration.
- Now references RFC 6819.
- Recorded more contributors.

-01

- Minor editorial changes.

-00

- Initial version.

8. References

8.1. Normative References

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

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

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

8.2. Informative References

[OpenID.Discovery] Sakimura, N., Bradley, J., Jones, M., and E. Jay, "OpenID Connect Discovery 1.0", May 2013.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", RFC 4949, August 2007.

Authors' Addresses

Nat Sakimura (editor)
Nomura Research Institute

Email: sakimura@gmail.com
URI: http://nat.sakimura.org/

John Bradley
Ping Identity

Email: jbradley@pingidentity.com

Naveen Agarwal
Google

Email: naa@google.com