Internet Engineering Task Force (IETF) N. Sakimura, Ed.

Request for Comments: 7636 Nomura Research Institute

Category: Standards Track J. Bradley

ISSN: 2070-1721 Ping Identity

N. Agarwal

Google

September 2015

Proof Key for Code Exchange by OAuth Public Clients

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 through the use of Proof Key for Code Exchange

(PKCE, pronounced "pixy").

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force

(IETF). It represents the consensus of the IETF community. It has

received public review and has been approved for publication by the

Internet Engineering Steering Group (IESG). Further information on

Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata,

and how to provide feedback on it may be obtained at

http://www.rfc-editor.org/info/rfc7636.

Copyright Notice

Copyright (c) 2015 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.

Sakimura, et al. Standards Track [Page 1]

RFC 7636 OAUTH PKCE September 2015

Table of Contents

1. Introduction ....................................................3

1.1. Protocol Flow ..............................................5

2. Notational Conventions ..........................................6

3. Terminology .....................................................7

3.1. Abbreviations ..............................................7

4. Protocol ........................................................8

4.1. Client Creates a Code Verifier .............................8

4.2. Client Creates the Code Challenge ..........................8

4.3. Client Sends the Code Challenge with the

Authorization Request ......................................9

4.4. Server Returns the Code ....................................9

4.4.1. Error Response ......................................9

4.5. Client Sends the Authorization Code and the Code

Verifier to the Token Endpoint ............................10

4.6. Server Verifies code\_verifier before Returning the

Tokens ....................................................10

5. Compatibility ..................................................11

6. IANA Considerations ............................................11

6.1. OAuth Parameters Registry .................................11

6.2. PKCE Code Challenge Method Registry .......................11

6.2.1. Registration Template ..............................12

6.2.2. Initial Registry Contents ..........................13

7. Security Considerations ........................................13

7.1. Entropy of the code\_verifier ..............................13

7.2. Protection against Eavesdroppers ..........................13

7.3. Salting the code\_challenge ................................14

7.4. OAuth Security Considerations .............................14

7.5. TLS Security Considerations ...............................15

8. References .....................................................15

8.1. Normative References ......................................15

8.2. Informative References ....................................16

Appendix A. Notes on Implementing Base64url Encoding without

Padding .............................................17

Appendix B. Example for the S256 code\_challenge\_method ...........17

Acknowledgements ..................................................19

Authors' Addresses ................................................20

Sakimura, et al. Standards Track [Page 2]

RFC 7636 OAUTH PKCE September 2015

1. Introduction

OAuth 2.0 [RFC6749] public clients are susceptible to the

authorization code interception attack.

In this attack, the attacker intercepts the authorization code

returned from the authorization endpoint within a communication path

not protected by Transport Layer Security (TLS), such as inter-

application communication within the client's operating system.

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

application running on the end device, such as a smartphone, issues

an OAuth 2.0 Authorization Request via the browser/operating system.

The Redirection Endpoint URI in this case typically uses a custom URI

scheme. Step (1) happens through a secure API that cannot be

intercepted, though it may potentially be observed in advanced attack

scenarios. The request then gets forwarded to the OAuth 2.0

authorization server in step (2). Because OAuth requires the use of

TLS, this communication is protected by TLS and cannot be

intercepted. The authorization server returns the authorization code

in step (3). In step (4), the Authorization Code is returned to the

requester via the Redirection Endpoint URI that was provided in step

(1).

Note that it is possible for a malicious app to register itself as a

handler for the custom scheme in addition to the legitimate OAuth 2.0

app. Once it does so, the malicious app is now able to intercept the

authorization code in step (4). This allows the attacker to request

and obtain an access token in steps (5) and (6), respectively.

Sakimura, et al. Standards Track [Page 3]

RFC 7636 OAUTH PKCE September 2015

+~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+

| End Device (e.g., Smartphone) |

| |

| +-------------+ +----------+ | (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 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 scheme to be registered by multiple applications.

2. The OAuth 2.0 authorization code grant is used.

3. The attacker has access to the OAuth 2.0 [RFC6749] "client\_id" and

"client\_secret" (if provisioned). All OAuth 2.0 native app

client-instances use the same "client\_id". Secrets provisioned in

client binary applications cannot be considered confidential.

4. Either one of the following condition is met:

4a. The attacker (via the installed application) is able to

observe only the responses from the authorization endpoint.

When "code\_challenge\_method" value is "plain", only this

attack is mitigated.

Sakimura, et al. Standards Track [Page 4]

RFC 7636 OAUTH PKCE September 2015

4b. A more sophisticated attack scenario allows the attacker 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. This was caused by leaking http

log information in the OS. To mitigate this,

"code\_challenge\_method" value must be set either to "S256" or

a value defined by a cryptographically secure

"code\_challenge\_method" extension.

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 the OAuth 2.0 threat model (Section 4.4.1 of

[RFC6819]) describes mitigation techniques, they are, unfortunately,

not applicable since they rely on a per-client instance secret or a

per-client instance redirect URI.

To mitigate this attack, this extension utilizes a dynamically

created cryptographically random key called "code verifier". A

unique 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, since it is sent

over TLS and cannot be intercepted.

1.1. Protocol Flow

+-------------------+

| Authz Server |

+--------+ | +---------------+ |

| |--(A)- Authorization Request ---->| | |

| | + t(code\_verifier), t\_m | | Authorization | |

| | | | Endpoint | |

| |<-(B)---- Authorization Code -----| | |

| | | +---------------+ |

| Client | | |

| | | +---------------+ |

| |--(C)-- Access Token Request ---->| | |

| | + code\_verifier | | Token | |

| | | | Endpoint | |

| |<-(D)------ Access Token ---------| | |

+--------+ | +---------------+ |

+-------------------+

Figure 2: Abstract Protocol Flow

Sakimura, et al. Standards Track [Page 5]

RFC 7636 OAUTH PKCE September 2015

This specification adds additional parameters to the OAuth 2.0

Authorization and Access Token Requests, shown in abstract form in

Figure 2.

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\_m".

B. The Authorization Endpoint responds as usual but records

"t(code\_verifier)" and the transformation method.

C. The client then sends the authorization code in 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 code 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 natural language meanings.

This specification uses the Augmented Backus-Naur Form (ABNF)

notation of [RFC5234].

STRING denotes a sequence of zero or more ASCII [RFC20] characters.

OCTETS denotes a sequence of zero or more octets.

ASCII(STRING) denotes the octets of the ASCII [RFC20] representation

of STRING where STRING is a sequence of zero or more ASCII

characters.

BASE64URL-ENCODE(OCTETS) denotes the base64url encoding of OCTETS,

per Appendix A, producing a STRING.

Sakimura, et al. Standards Track [Page 6]

RFC 7636 OAUTH PKCE September 2015

BASE64URL-DECODE(STRING) denotes the base64url decoding of STRING,

per Appendix A, producing a sequence of octets.

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

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.

code challenge method

A method that was used to derive code challenge.

Base64url Encoding

Base64 encoding using the URL- and filename-safe character set

defined in Section 5 of [RFC4648], with all trailing '='

characters omitted (as permitted by Section 3.2 of [RFC4648]) and

without the inclusion of any line breaks, whitespace, or other

additional characters. (See Appendix A for notes on implementing

base64url encoding without padding.)

3.1. Abbreviations

ABNF Augmented Backus-Naur Form

Authz Authorization

PKCE Proof Key for Code Exchange

MITM Man-in-the-middle

MTI Mandatory To Implement

Sakimura, et al. Standards Track [Page 7]

RFC 7636 OAUTH PKCE September 2015

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 STRING using the

unreserved characters [A-Z] / [a-z] / [0-9] / "-" / "." / "\_" / "~"

from Section 2.3 of [RFC3986], with a minimum length of 43 characters

and a maximum length of 128 characters.

ABNF for "code\_verifier" is as follows.

code-verifier = 43\*128unreserved

unreserved = ALPHA / DIGIT / "-" / "." / "\_" / "~"

ALPHA = %x41-5A / %x61-7A

DIGIT = %x30-39

NOTE: The 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

43-octet URL safe string to use as the code verifier.

4.2. Client Creates the Code Challenge

The client then creates a 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-ENCODE(SHA256(ASCII(code\_verifier)))

If the client is capable of using "S256", it MUST use "S256", as

"S256" is Mandatory To Implement (MTI) on the server. Clients are

permitted to use "plain" only if they cannot support "S256" for some

technical reason and know via out-of-band configuration that the

server supports "plain".

The plain transformation is for compatibility with existing

deployments and for constrained environments that can't use the S256

transformation.

Sakimura, et al. Standards Track [Page 8]

RFC 7636 OAUTH PKCE September 2015

ABNF for "code\_challenge" is as follows.

code-challenge = 43\*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

Authorization Request (Section 4.1.1 of [RFC6749]) using the

following additional parameters:

code\_challenge

REQUIRED. Code challenge.

code\_challenge\_method

OPTIONAL, defaults to "plain" if not present in the request. Code

verifier transformation method is "S256" or "plain".

4.4. Server Returns the Code

When the server issues the authorization code in the authorization

response, it MUST associate the "code\_challenge" and

"code\_challenge\_method" values with the authorization 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 Proof Key for Code Exchange (PKCE) by OAuth

public clients and the client does not send the "code\_challenge" in

the request, the authorization endpoint MUST return the authorization

error response with the "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.

Sakimura, et al. Standards Track [Page 9]

RFC 7636 OAUTH PKCE September 2015

If the server supporting PKCE does not support the requested

transformation, 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.

4.5. Client Sends the Authorization Code and the Code Verifier to the

Token Endpoint

Upon receipt of the Authorization Code, the client sends the Access

Token Request to the token endpoint. In addition to the parameters

defined in the OAuth 2.0 Access Token Request (Section 4.1.3 of

[RFC6749]), it sends the following parameter:

code\_verifier

REQUIRED. Code verifier

The "code\_challenge\_method" is bound to the Authorization Code when

the Authorization Code is issued. That is the method that the token

endpoint MUST use to verify the "code\_verifier".

4.6. 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 the 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 4.3 was "S256", the

received "code\_verifier" is hashed by SHA-256, base64url-encoded, and

then compared to the "code\_challenge", i.e.:

BASE64URL-ENCODE(SHA256(ASCII(code\_verifier))) == code\_challenge

If the "code\_challenge\_method" from Section 4.3 was "plain", they are

compared directly, i.e.:

code\_verifier == code\_challenge.

If the values are equal, the 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 [RFC6749] MUST be returned.

Sakimura, et al. Standards Track [Page 10]

RFC 7636 OAUTH PKCE September 2015

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 revert to the OAuth 2.0 [RFC6749]

protocol without this extension.

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 4 to

all servers.

6. IANA Considerations

IANA has made the following registrations per this document.

6.1. OAuth Parameters Registry

This specification registers the following parameters in the IANA

"OAuth Parameters" registry defined in OAuth 2.0 [RFC6749].

o Parameter name: code\_verifier

o Parameter usage location: token request

o Change controller: IESG

o Specification document(s): RFC 7636 (this document)

o Parameter name: code\_challenge

o Parameter usage location: authorization request

o Change controller: IESG

o Specification document(s): RFC 7636 (this document)

o Parameter name: code\_challenge\_method

o Parameter usage location: authorization request

o Change controller: IESG

o Specification document(s): RFC 7636 (this document)

6.2. PKCE Code Challenge Method Registry

This specification establishes the "PKCE Code Challenge Methods"

registry. The new registry should be a sub-registry of the "OAuth

Parameters" registry.

Additional "code\_challenge\_method" types for use with the

authorization endpoint are registered using the Specification

Required policy [RFC5226], which includes review of the request by

one or more Designated Experts (DEs). The DEs will ensure that there

Sakimura, et al. Standards Track [Page 11]

RFC 7636 OAUTH PKCE September 2015

is at least a two-week review of the request on the oauth-ext-

review@ietf.org mailing list and that any discussion on that list

converges before they respond to the request. To allow for the

allocation of values prior to publication, the Designated Expert(s)

may approve registration once they are satisfied that an acceptable

specification will be published.

Registration requests and discussion on the oauth-ext-review@ietf.org

mailing list should use an appropriate subject, such as "Request for

PKCE code\_challenge\_method: example").

The Designated Expert(s) should consider the discussion on the

mailing list, as well as the overall security properties of the

challenge method when evaluating registration requests. New methods

should not disclose the value of the code\_verifier in the request to

the Authorization endpoint. Denials should include an explanation

and, if applicable, suggestions as to how to make the request

successful.

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) states

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, and home page URI) may also be included.

Specification Document(s):

Reference to the document(s) that specifies 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.

Sakimura, et al. Standards Track [Page 12]

RFC 7636 OAUTH PKCE September 2015

6.2.2. Initial Registry Contents

Per this document, IANA has registered the Code Challenge Method

Parameter Names defined in Section 4.2 in this registry.

o Code Challenge Method Parameter Name: plain

o Change Controller: IESG

o Specification Document(s): Section 4.2 of RFC 7636 (this document)

o Code Challenge Method Parameter Name: S256

o Change Controller: IESG

o Specification Document(s): Section 4.2 of RFC 7636 (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.

The client SHOULD create a "code\_verifier" with a minimum of 256 bits

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 43-octet URL safe string to use as a

"code\_challenge" that has the required entropy.

7.2. Protection against Eavesdroppers

Clients MUST NOT downgrade to "plain" after trying the "S256" method.

Servers that support PKCE are required to support "S256", and servers

that do not support PKCE will simply ignore the unknown

"code\_verifier". Because of this, an error when "S256" is presented

can only mean that the server is faulty or that a MITM attacker is

trying a downgrade attack.

The "S256" method protects against eavesdroppers observing or

intercepting the "code\_challenge", because the challenge cannot be

used without the verifier. With the "plain" method, there is a

chance that "code\_challenge" will be observed by the attacker on the

device or in the http request. Since the code challenge is the same

as the code verifier in this case, the "plain" method does not

protect against the eavesdropping of the initial request.

The use of "S256" protects against disclosure of the "code\_verifier"

value to an attacker.

Sakimura, et al. Standards Track [Page 13]

RFC 7636 OAUTH PKCE September 2015

Because of this, "plain" SHOULD NOT be used and exists only for

compatibility with deployed implementations where the request path is

already protected. The "plain" method SHOULD NOT be used in new

implementations, unless they cannot support "S256" for some technical

reason.

The "S256" code challenge method or other cryptographically secure

code challenge method extension SHOULD be used. The "plain" code

challenge method relies on the operating system and transport

security not to disclose the request to an attacker.

If the code challenge method is "plain" and the code challenge is to

be returned inside authorization "code" to achieve a stateless

server, it MUST be encrypted in such a manner that only the server

can decrypt and extract it.

7.3. Salting the code\_challenge

To reduce implementation complexity, salting is not used in the

production of the code challenge, as the code verifier contains

sufficient entropy to prevent brute-force attacks. Concatenating a

publicly known value to a code verifier (containing 256 bits of

entropy) and then hashing it with SHA256 to produce a code challenge

would not increase the number of attempts necessary to brute force a

valid value for code verifier.

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.

7.4. OAuth Security Considerations

All the OAuth security analysis presented in [RFC6819] applies, so

readers SHOULD carefully follow it.

Sakimura, et al. Standards Track [Page 14]

RFC 7636 OAUTH PKCE September 2015

7.5. TLS Security Considerations

Current security considerations can be found in "Recommendations for

Secure Use of Transport Layer Security (TLS) and Datagram Transport

Layer Security (DTLS)" [BCP195]. This supersedes the TLS version

recommendations in OAuth 2.0 [RFC6749].

8. References

8.1. Normative References

[BCP195] Sheffer, Y., Holz, R., and P. Saint-Andre,

"Recommendations for Secure Use of Transport Layer

Security (TLS) and Datagram Transport Layer Security

(DTLS)", BCP 195, RFC 7525, May 2015,

<http://www.rfc-editor.org/info/bcp195>.

[RFC20] Cerf, V., "ASCII format for network interchange", STD 80,

RFC 20, DOI 10.17487/RFC0020, October 1969,

<http://www.rfc-editor.org/info/rfc20>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate

Requirement Levels", BCP 14, RFC 2119,

DOI 10.17487/RFC2119, March 1997,

<http://www.rfc-editor.org/info/rfc2119>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform

Resource Identifier (URI): Generic Syntax", STD 66, RFC

3986, DOI 10.17487/RFC3986, January 2005,

<http://www.rfc-editor.org/info/rfc3986>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data

Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006,

<http://www.rfc-editor.org/info/rfc4648>.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an

IANA Considerations Section in RFCs", BCP 26, RFC 5226,

DOI 10.17487/RFC5226, May 2008,

<http://www.rfc-editor.org/info/rfc5226>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax

Specifications: ABNF", STD 68, RFC 5234,

DOI 10.17487/RFC5234, January 2008,

<http://www.rfc-editor.org/info/rfc5234>.

Sakimura, et al. Standards Track [Page 15]

RFC 7636 OAUTH PKCE September 2015

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms

(SHA and SHA-based HMAC and HKDF)", RFC 6234,

DOI 10.17487/RFC6234, May 2011,

<http://www.rfc-editor.org/info/rfc6234>.

[RFC6749] Hardt, D., Ed., "The OAuth 2.0 Authorization Framework",

RFC 6749, DOI 10.17487/RFC6749, October 2012,

<http://www.rfc-editor.org/info/rfc6749>.

8.2. Informative References

[RFC6819] Lodderstedt, T., Ed., McGloin, M., and P. Hunt, "OAuth 2.0

Threat Model and Security Considerations", RFC 6819,

DOI 10.17487/RFC6819, January 2013,

<http://www.rfc-editor.org/info/rfc6819>.

Sakimura, et al. Standards Track [Page 16]

RFC 7636 OAUTH PKCE September 2015

Appendix A. Notes on Implementing Base64url Encoding without Padding

This appendix describes how to implement a base64url-encoding

function without padding, based upon the standard base64-encoding

function that uses 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;

}

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

Appendix B. Example for the S256 code\_challenge\_method

The client uses output of a suitable random number generator to

create a 32-octet sequence. The octets representing the value in

this example (using JSON array notation) are:

[116, 24, 223, 180, 151, 153, 224, 37, 79, 250, 96, 125, 216, 173,

187, 186, 22, 212, 37, 77, 105, 214, 191, 240, 91, 88, 5, 88, 83,

132, 141, 121]

Encoding this octet sequence as base64url provides the value of the

code\_verifier:

dBjftJeZ4CVP-mB92K27uhbUJU1p1r\_wW1gFWFOEjXk

The code\_verifier is then hashed via the SHA256 hash function to

produce:

[19, 211, 30, 150, 26, 26, 216, 236, 47, 22, 177, 12, 76, 152, 46,

8, 118, 168, 120, 173, 109, 241, 68, 86, 110, 225, 137, 74, 203,

112, 249, 195]

Sakimura, et al. Standards Track [Page 17]

RFC 7636 OAUTH PKCE September 2015

Encoding this octet sequence as base64url provides the value of the

code\_challenge:

E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM

The authorization request includes:

code\_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM

&code\_challenge\_method=S256

The authorization server then records the code\_challenge and

code\_challenge\_method along with the code that is granted to the

client.

In the request to the token\_endpoint, the client includes the code

received in the authorization response as well as the additional

parameter:

code\_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r\_wW1gFWFOEjXk

The authorization server retrieves the information for the code

grant. Based on the recorded code\_challenge\_method being S256, it

then hashes and base64url-encodes the value of code\_verifier:

BASE64URL-ENCODE(SHA256(ASCII(code\_verifier)))

The calculated value is then compared with the value of

"code\_challenge":

BASE64URL-ENCODE(SHA256(ASCII(code\_verifier))) == code\_challenge

If the two values are equal, then the authorization server can

provide the tokens as long as there are no other errors in the

request. If the values are not equal, then the request must be

rejected, and an error returned.

Sakimura, et al. Standards Track [Page 18]

RFC 7636 OAUTH PKCE September 2015

Acknowledgements

The initial draft version of this specification was created by the

OpenID AB/Connect Working Group of the OpenID Foundation.

This specification is the work of the OAuth Working Group, which

includes dozens of active and dedicated participants. In particular,

the following individuals contributed ideas, feedback, and wording

that shaped and formed the final specification:

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

Justin Richer, MIT Kerberos

Josh Mandel, Boston Children's Hospital

Lewis Adam, Motorola Solutions

Madjid Nakhjiri, Samsung

Michael B. Jones, Microsoft

Paul Madsen, Ping Identity

Phil Hunt, Oracle

Prateek Mishra, Oracle

Ryo Ito, mixi

Scott Tomilson, Ping Identity

Sergey Beryozkin

Takamichi Saito

Torsten Lodderstedt, Deutsche Telekom

William Denniss, Google

Sakimura, et al. Standards Track [Page 19]

RFC 7636 OAUTH PKCE September 2015

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 Parkway

Mountain View, CA 94043

United States

Phone: +1 650-253-0000

Email: naa@google.com

URI: http://google.com/

Sakimura, et al. Standards Track [Page 20]