Proof Key for Code Exchange by OAuth Public Clients
draft-ietf-oauth-spop-15

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 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 January 9, 2016.

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 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 also 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).

A malicious app that has been designed to attack this native app has
previously registered itself as a handler for the custom URI scheme
is now able to intercept the Authorization Code in step (4). This
allows the attacker to request and obtain an access token in steps
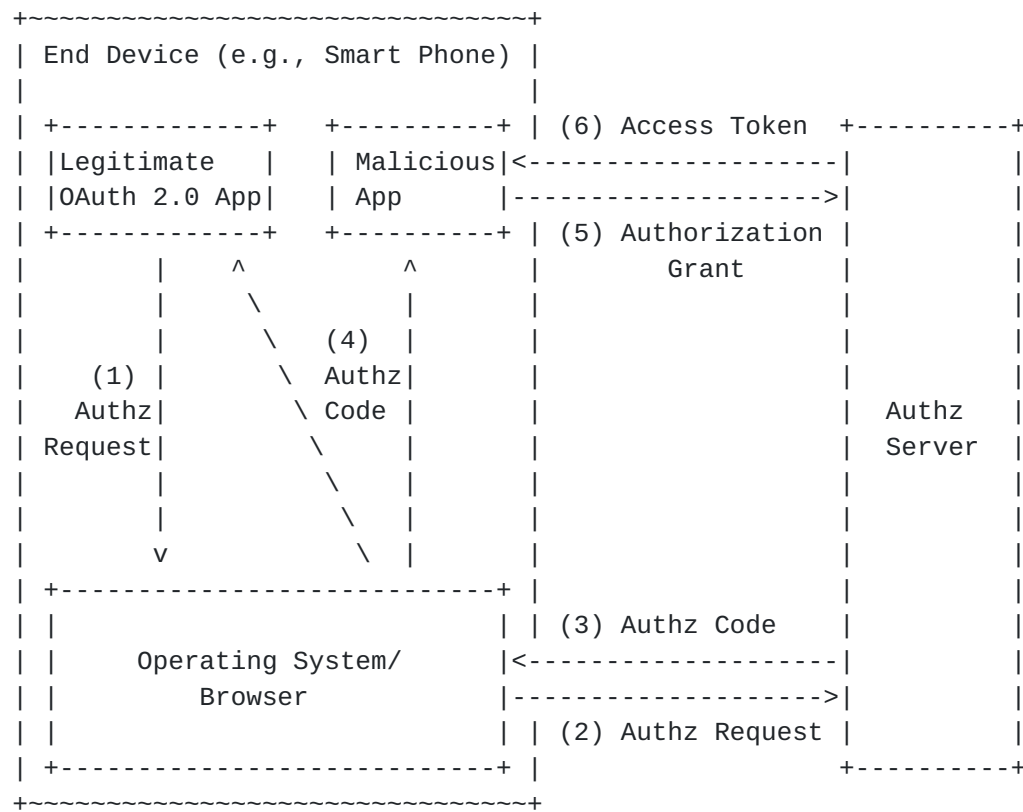(5) and (6), respectively.

```
 +~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~+
 | 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 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.
4a) The attacker (via the installed app) is able to observe only the responses from the authorization endpoint. The plain code_challenge_method mitigates only this attack.
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 has been caused by leaking http log information in the OS. To mitigate this the S256 code_challenge_method or

cryptographically secure code_challenge_method extension must be
used.

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 [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". 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 --------|          | |
        +--------+                       | +---------------+ |
                                         +-------------------+
```
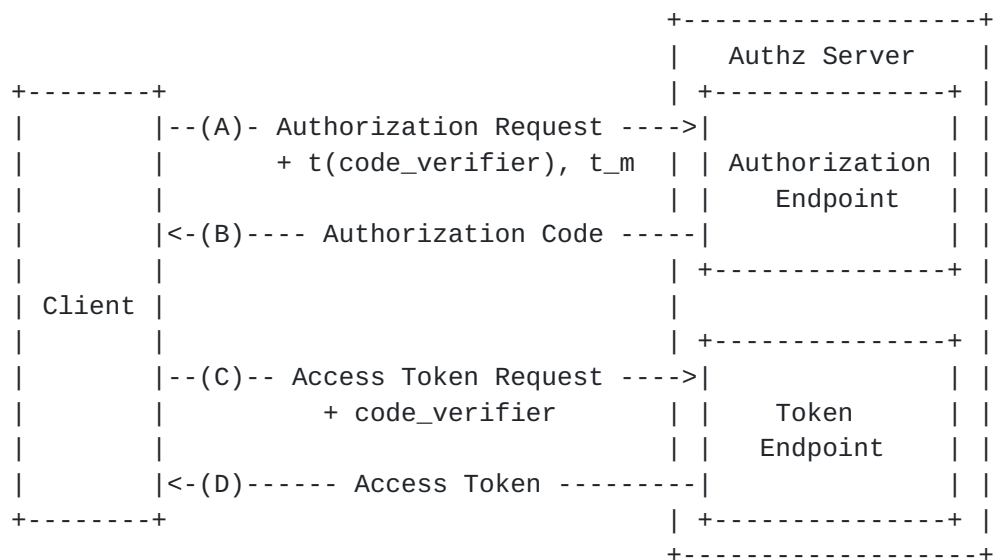
Figure 2: Abstract Protocol Flow

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_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 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].

STRING denotes a sequence of zero or more ASCII [RFC0020] characters.

OCTETS denotes a sequence of zero or more octets.

ASCII(STRING) denotes the octets of the ASCII [RFC0020]
representation of STRING where STRING is a sequence of zero or more
ASCII characters.

BASE64URL-ENCODE(OCTETS) denotes the base64url encoding of OCTETS,
per Section 3 producing a STRING.

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

SHA256(OCTETS) denotes a SHA2 256bit 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.
   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

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 Sec 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: 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.

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, "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 "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.

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 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 hashed by SHA-256, then 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 4.2 was "plain", they are
   compared directly. i.e.,

   "code_verifier" == "code_challenge".

   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 [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 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].

   o Parameter name: code_verifier

   o Parameter usage location: token request
   o Change controller: IESG
   o Specification document(s): this document

   o Parameter name: code_challenge
   o Parameter usage location: authorization request
   o Change controller: IESG
   o Specification document(s): this document

   o Parameter name: code_challenge_method
   o Parameter usage location: authorization request
   o Change controller: IESG
   o Specification document(s): this document

6.2. PKCE Code Challenge Method Registry

   This specification establishes the PKCE Code Challenge Method
   registry. The new registry should be a sub-registry of 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. The DEs will ensure there 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) 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.

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

o Code Challenge Method Parameter Name: "S256"
o Change Controller: IESG
o 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.

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 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 "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" OAuth 2.0 (see Section 3.2 of [RFC6749]. Because of
that, an error when "S256" is presented can only mean that the server
is faulty or that a MITM attacker is trying a downgrade attack.

"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, "plain" method does not protect
against the eavesdropping of the initial request.

The use of "S256" protects against disclosure of "code_verifier"
value to an attacker.

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

In order 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.

7.5. TLS security considerations

Curent security considerations can be found in Recommendations for
Secure Use of TLS and DTLS [BCP195]. This supersedes the TLS version
recommendations in OAuth 2.0 [RFC6749].

8. Acknowledgements

The initial draft 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
    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 Tomilson, Ping Identity
        Sergey Beryozkin
        Takamichi Saito
        Torsten Lodderstedt, Deutsche Telekom
        William Denniss, Google

9. Revision History

    -15

    o Addressed Barry's IESG comments around IANA Registration
    o Addressed Barry's IESG comments around Sec 7.2 downgrade attack
    o fix a typo for William and make a small change to Fig 1.1
       clarifying t_m
    o more wording changes to sec 7.2 re Barry
    o made the two SHOULD NOT use plain recommendations consistent.
    o slightly cleaned up grammer in Sec 7.2

    -14

    o #38. Expanded Section 7.2 to explain why plain should not be
       used.
    o #39. Modified Section 4.4.1 to discourage the use of plain.
    o #40. Modified Intro text to explain the attack better.
    o #41. Added explanation that the token request is protected in the
       Last paragraph of the Introduction.
    o #42. Sec 4.2: Removed redundant double quotes caused by spanx.
    o #43. Sec 4.4: Replaced code with authorization code.
    o #44. Sec 4.5: say "code_verifier" rather than "secret"
    o #45. Sec 4.4.1: Expanded PKCE.
    o #46. Sec 5: SHOULD in para 1 removed.
    o Added abbreviations section.

    -13

    o Fix the parameter usage locations for the OAuth Parameters
       Registry per Hannes response.
    o Clarify for IANA that the new registry is a sub-registry of OAuth
       Parameters registry
    o aded text on why the code_challenge_method is not sent to the
       token endpoint.

    -12

o clarify that the client secret we are talking about in the
   Introduction is a OAuth 2 client_secret.
o Update salting security consideration based on Ben's feedback

-11

o add spanx for plain in sec 4.4 RE Kathleen's comment
o Add security consideration on TLS and reference BCP195
o Update to make clearer that plain can only be used for backwards
   compatibility and constrained environments

-10

o re #33 specify lower limit to code_verifier in prose
o remove base64url decode from draft, all steps now use encode only
o Expanded MTI
o re #33 change length of 32 octet base64url encoded string back to
   43 octets

-09

o clean up some external references so they don't point at internal
   sections

-08

o changed BASE64URL to BASE64URL-ENCODE to be more consistent with
   appendix A Fixed lowercase base64url in appendix B
o Added appendix B as an example of S256 processing
o Change reference for unreserved characters to RFC3986 from
   base64URL

-07

o removed unused discovery reference and UTF8
o re #32 added ASCII(STRING) to make clear that it is the byte array
   that is being hashed
o re #2 Remove discovery requirement section.
o updated Acknowledgement
o re #32 remove unneeded UTF8(STRING) definition, and define STRING
   for ASCII(STRING)
o re #32 remove unneeded utf8 reference from BASE64URL-
   DECODE(STRING) def
o resolves #31 unused definition of concatenation
o re #30 Update figure text call out the endpoints
o re #30 Update figure to call out the endpoints
o small wording change to the introduction

   -06

   o fix date
   o replace spop with pkce for registry and other references
   o re #29 change name again
   o re #27 removed US-ASCII reference
   o re #27 updated ABNF for code_verifier
   o resolves #24 added security consideration for salting
   o resolves #29 Changed title
   o updated reference to RFC4634 to RFC6234 re #27
   o changed reference for US-ASCII to RFC20 re #27
   o resolves #28 added Acknowledgements
   o resolves #27 updated ABNF
   o resolves #26 updated abstract and added Hannes figure

   -05

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

   -04

   o Added error response to authorization response.

   -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

   [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.

   [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.

   [RFC3986]   Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform
               Resource Identifier (URI): Generic Syntax", STD 66, RFC
               3986, January 2005.

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

   [RFC5226]   Narten, T. and H. Alvestrand, "Guidelines for Writing an
               IANA Considerations Section in RFCs", BCP 26, RFC 5226,
               May 2008.

   [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

   [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 a base64url encoding
   function without padding based upon 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 a 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]

Encoding this octet sequence as a base64url provides the value of the
code_challenge:

    E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM

The authorization request includes:

```
        code_challenge=E9Melhoa2OwvFrEMTJguCHaoeK1t8URWbuGJSstw-cM
        &code_challange_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
paramater:

```
        code_verifier=dBjftJeZ4CVP-mB92K27uhbUJU1p1r_wW1gFWFOEjXk
```

The Authorization server retrieves the information for the code
grant. Based on the recorded code_challange_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.

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/