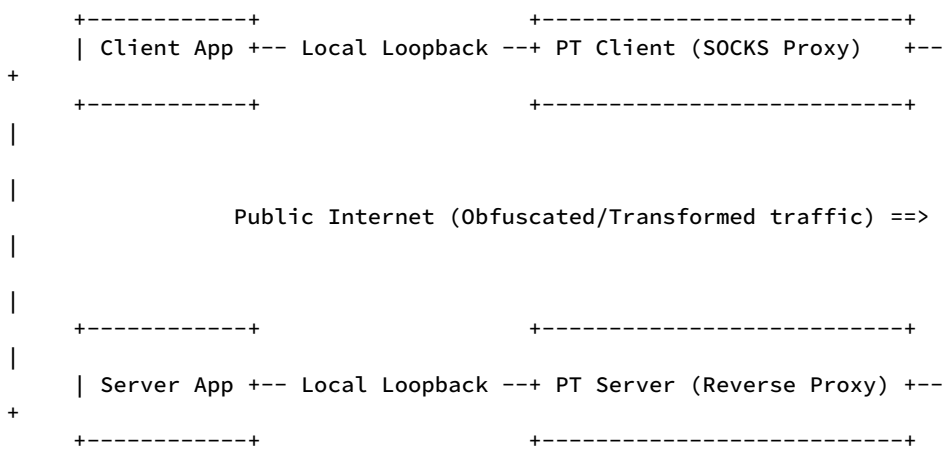
Architecture Overview



On the client's host, the PT Client software exposes a SOCKS proxy [RFC1928] to the client application, and obfuscates or otherwise transforms traffic before forwarding it to the server's host.

On the server's host, the PT Server software exposes a reverse proxy that accepts connections from PT Clients, and handles reversing the obfuscation/ transformation applied to traffic, before forwarding it to the actual server software. An optional lightweight protocol exists to facilitate communicating connection meta-data that would otherwise be lost such as the source IP address and port [EXTORPORT].

All PT instances are configured by the respective parent process via a set of standardized environment variables (3.2) that are set at launch time, and report status information back to the parent via writing output in a standardized format to stdout (3.3).

Each invocation of a PT MUST be either a client OR a server.

All PT client forward proxies MUST support either SOCKS 4 or SOCKS 5, and SHOULD prefer SOCKS 5 over SOCKS 4.

This is for the rate limiting DoS mitigation specifically.

The `EXT_FIELD_TYPE` value for the `DOS_PARAMS` extension is `[01]`.

The content is defined as follows:

Field	Size	Description
`N_PARAMS`	1	Number of parameters
`N_PARAMS` times:		
- PARAM_TYPE	1	Identifier for a parameter
- PARAM_VALUE	8	Integer value

Recognized values for `PARAM_TYPE` in this extension are:

PARAM_TYPE	Name	Min	Max
`[01]`	`DOS_INTRODUCE2_RATE_PER_SEC`	0	0x7fffffff
`[02]`	`DOS_INTRODUCE2_BURST_PER_SEC`	0	0x7fffffff

Together, these parameters configure a token bucket that determines how many INTRODUCE2 messages the introduction point may send to the service.

The `DOS_INTRODUCE2_RATE_PER_SEC` parameter defines the maximum average rate of messages; The `DOS_INTRODUCE2_BURST_PER_SEC` parameter defines the largest allowable burst of messages (that is, the size of the token bucket).

> Technically speaking, the `BURST` parameter is misnamed in that it is not actually "per second": only a *rate* has an associated time.

If either of these parameters is set to 0, the defense is disabled, and the introduction point should ignore the other parameter.

If the burst is lower than the rate, the introduction point SHOULD ignore the extension.

> Using this extension extends the body of the ESTABLISH_INTRO message by 19 bytes bringing it from 134 bytes to 155 bytes.

When this extension is not *sent*, introduction points use default settings taken from taken from the consensus parameters HiddenServiceEnableIntroDoSDefense, HiddenServiceEnableIntroDoSRatePerSec, and HiddenServiceEnableIntroDoSBurstPerSec.

This extension can only be used with relays supporting the subprotocol "HSIntro=5".

Introduced in tor-0.4.2.1-alpha.

Registering an introduction point on a legacy Tor node

> This section is obsolete and refers to a workaround for now-obsolete Tor relay versions. It is included for historical reasons.

Tor nodes should also support an older version of the ESTABLISH_INTRO message, first documented in rend-spec.txt. New hidden service hosts must use this format when establishing introduction points at older Tor nodes that do not support the format above in [EST_INTRO].

In this older protocol, an ESTABLISH_INTRO message contains:

```
      KEY_LEN        [2 bytes]
      KEY            [KEY_LEN bytes]
      HANDSHAKE_AUTH [20 bytes]
      SIG            [variable, up to end of relay message body]

  The KEY_LEN variable determines the length of the KEY field.
```

The KEY field is the ASN1-encoded legacy RSA public key that was also included in the hidden service descriptor.

The HANDSHAKE_AUTH field contains the SHA1 digest of (KH | "INTRODUCE").

The SIG field contains an RSA signature, using PKCS1 padding, of all earlier fields.

Older versions of Tor always use a 1024-bit RSA key for these introduction authentication keys.

Acknowledging establishment of introduction point

After setting up an introduction circuit, the introduction point reports its status back to the hidden service host with an INTRO_ESTABLISHED message.

The INTRO_ESTABLISHED message has the following contents:

Introduction

This specification describes a way to decouple protocol-level obfuscation from an application's client/server code, in a manner that promotes rapid development of obfuscation/circumvention tools and promotes reuse beyond the scope of the Tor Project's efforts in that area.

This is accomplished by utilizing helper sub-processes that implement the necessary forward/reverse proxy servers that handle the censorship circumvention, with a well defined and standardized configuration and management interface.

Any application code that implements the interfaces as specified in this document will be able to use all spec compliant Pluggable Transports.

Requirements Notation

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 [RFC2119].

Pluggable Transport Specification (Version 1)

Abstract

Pluggable Transports (PTs) are a generic mechanism for the rapid development and deployment of censorship circumvention, based around the idea of modular sub-processes that transform traffic to defeat censors.

This document specifies the sub-process startup, shutdown, and inter-process communication mechanisms required to utilize PTs.

```
      N_EXTENSIONS [1 byte]
      N_EXTENSIONS times:
        EXT_FIELD_TYPE [1 byte]
        EXT_FIELD_LEN  [1 byte]
        EXT_FIELD      [EXT_FIELD_LEN bytes]
```

Older versions of Tor send back an empty INTRO_ESTABLISHED message instead. Services must accept an empty INTRO_ESTABLISHED message from a legacy relay. [The above paragraph is obsolete and refers to a workaround for now-obsolete Tor relay versions. It is included for historical reasons.]

The same rules for multiplicity, ordering, and handling unknown types apply to the extension fields here as described [EST_INTRO] above.

Sending an INTRODUCE1 message to the introduction point

In order to participate in the introduction protocol, a client must know the following:

```
      * An introduction point for a service.
      * The introduction authentication key for that introduction
 point.
      * The introduction encryption key for that introduction point.
```

The client sends an INTRODUCE1 message to the introduction point, containing an identifier for the service, an identifier for the encryption key that the client intends to use, and an opaque blob to be relayed to the hidden service host.

In reply, the introduction point sends an INTRODUCE_ACK message back to the client, either informing it that its request has been delivered, or that its request will not succeed.

If the INTRODUCE_ACK message indicates success, the client SHOULD close the circuit to the introduction point, and not use it for anything else. If the INTRODUCE_ACK message indicates failure, the client MAY try a different introduction point. It MAY reach the different introduction point either by extending its introduction circuit an additional hop, or by building a new introduction circuit.

```
    [TODO: specify what tor should do when receiving a malformed
 message. Drop it?
           Kill circuit? This goes for all possible messages.]
```

Extensible INTRODUCE1 message format

When a client is connecting to an introduction point, INTRODUCE1 messages should be of the form:

```
LEGACY_KEY_ID   [20 bytes]
AUTH_KEY_TYPE   [1 byte]
AUTH_KEY_LEN    [2 bytes]
AUTH_KEY        [AUTH_KEY_LEN bytes]
N_EXTENSIONS    [1 byte]
N_EXTENSIONS times:
  EXT_FIELD_TYPE [1 byte]
  EXT_FIELD_LEN  [1 byte]
  EXT_FIELD      [EXT_FIELD_LEN bytes]
ENCRYPTED        [Up to end of relay message body]
```

The `ENCRYPTED` field is described in the [PROCESS_INTRO2] section.

AUTH_KEY_TYPE is defined as in [EST_INTRO]. Currently, the only value of AUTH_KEY_TYPE for this message is an Ed25519 public key [02].

The LEGACY_KEY_ID field is used to distinguish between legacy and new style INTRODUCE1 messages. In new style INTRODUCE1 messages, LEGACY_KEY_ID is 20 zero bytes. Upon receiving an INTRODUCE1 messages, the introduction point checks the LEGACY_KEY_ID field. If LEGACY_KEY_ID is non-zero, the INTRODUCE1 message should be handled as a legacy INTRODUCE1 message by the intro point.

Upon receiving a INTRODUCE1 message, the introduction point checks whether AUTH_KEY matches the introduction point authentication key for an active introduction circuit. If so, the introduction point sends an INTRODUCE2 message with exactly the same contents to the service, and sends an INTRODUCE_ACK response to the client.

(Note that the introduction point does not "clean up" the INTRODUCE1 message that it retransmits. Specifically, it does not change the order or multiplicity of the extensions sent by the client.)

The same rules for multiplicity, ordering, and handling unknown types apply to the extension fields here as described [EST_INTRO] above.

INTRODUCE_ACK message format.

An INTRODUCE_ACK message has the following fields:

USERADDR

```
An ASCII string holding the TCP/IP address of the client of the
pluggable transport proxy. A Tor bridge SHOULD use that address to
collect statistics about its clients.  Recognized formats are:
  1.2.3.4:5678
  [1:2::3:4]:5678
```

(Current Tor versions may accept other formats, but this is a bug: transports MUST NOT send them.)

The string MUST not be NUL-terminated.

TRANSPORT

An ASCII string holding the name of the pluggable transport used by the client of the pluggable transport proxy. A Tor bridge that supports multiple transports SHOULD use that information to collect statistics about the popularity of individual pluggable transports.

The string MUST not be NUL-terminated.

Pluggable transport names are C-identifiers and Tor MUST check them for correctness.

Security Considerations

Extended ORPort or TransportControlPort do *not* provide link confidentiality, authentication or integrity. Sensitive data, like cryptographic material, should not be transferred through them.

An attacker with superuser access is able to sniff network traffic, and capture TransportControlPort identifiers and any data passed through those ports.

Tor SHOULD issue a warning if the bridge operator tries to bind Extended ORPort to a non-localhost address.

Pluggable transport proxies SHOULD issue a warning if they are instructed to connect to a non-localhost Extended ORPort.

The extended ORPort protocol

Once a connection is established and authenticated, the parties communicate with the protocol described here.

Protocol

The extended server port protocol is as follows:

```
      COMMAND [2 bytes, big-endian]
      BODYLEN [2 bytes, big-endian]
      BODY [BODYLEN bytes]

      Commands sent from the transport proxy to the bridge are:

      [0x0000] DONE: There is no more information to give. The next
        bytes sent by the transport will be those tunneled over it.
        (body ignored)

      [0x0001] USERADDR: an address:port string that represents the
        client's address.

      [0x0002] TRANSPORT: a string of the name of the pluggable
        transport currently in effect on the connection.

      Replies sent from tor to the proxy are:

      [0x1000] OKAY: Send the user's traffic. (body ignored)

      [0x1001] DENY: Tor would prefer not to get more traffic from
        this address for a while. (body ignored)

      [0x1002] CONTROL: (Not used)

   Parties MUST ignore command codes that they do not understand.
```

If the server receives a recognized command that does not parse, it MUST close the connection to the client.

Command descriptions

```
      STATUS       [2 bytes]
      N_EXTENSIONS [1 bytes]
      N_EXTENSIONS times:
        EXT_FIELD_TYPE [1 byte]
        EXT_FIELD_LEN  [1 byte]
        EXT_FIELD      [EXT_FIELD_LEN bytes]

   Recognized status values are:

      [00 00] -- Success: message relayed to hidden service host.
      [00 01] -- Failure: service ID not recognized
      [00 02] -- Bad message format
      [00 03] -- Can't relay message to service
```

The same rules for multiplicity, ordering, and handling unknown types apply to the extension fields here as described [EST_INTRO] above.

Processing an INTRODUCE2 message at the hidden service.

Upon receiving an INTRODUCE2 message, the hidden service host checks whether the AUTH_KEY or LEGACY_KEY_ID field matches the keys for this introduction circuit.

The service host then checks whether it has received a message with these contents or rendezvous cookie before. If it has, it silently drops it as a replay. (It must maintain a replay cache for as long as it accepts messages with the same encryption key. Note that the encryption format below should be non-malleable. See also More notes on replay resistance below.)

If the message is not a replay, it decrypts the ENCRYPTED field, establishes a shared key with the client, and authenticates the whole contents of the message as having been unmodified since they left the client. There may be multiple ways of decrypting the ENCRYPTED field, depending on the chosen type of the encryption key. Requirements for an introduction handshake protocol are described in [INTRO-HANDSHAKE-REQS]. We specify one below in section [NTOR-WITH-EXTRA-DATA].

The decrypted plaintext must have the form:

```
      RENDEZVOUS_COOKIE                           [20 bytes]
      N_EXTENSIONS                                [1 byte]
      N_EXTENSIONS times:
          EXT_FIELD_TYPE                          [1 byte]
          EXT_FIELD_LEN                           [1 byte]
          EXT_FIELD                               [EXT_FIELD_LEN bytes]
      ONION_KEY_TYPE                              [1 bytes]
      ONION_KEY_LEN                               [2 bytes]
      ONION_KEY                                   [ONION_KEY_LEN bytes]
      NSPEC      (Number of link specifiers)      [1 byte]
      NSPEC times:
          LSTYPE (Link specifier type)            [1 byte]
          LSLEN  (Link specifier length)          [1 byte]
          LSPEC  (Link specifier)                 [LSLEN bytes]
      PAD        (optional padding)               [up to end of
plaintext]
```

Upon processing this plaintext, the hidden service makes sure that any required authentication is present in the extension fields, and then extends a rendezvous circuit to the node described in the LSPEC fields, using the ONION_KEY to complete the extension. As mentioned in [BUILDING-BLOCKS], the “TLS-over-TCP, IPv4” and “Legacy node identity” specifiers must be present.

As of 0.4.1.1-alpha, clients include both IPv4 and IPv6 link specifiers in INTRODUCE1 messages. All available addresses SHOULD be included in the message, regardless of the address that the client actually used to extend to the rendezvous point.

The hidden service should handle invalid or unrecognised link specifiers the same way as clients do in section 2.5.2.2. In particular, services SHOULD perform basic validity checks on link specifiers, and SHOULD NOT reject unrecognised link specifiers, to avoid information leaks. The list of link specifiers received here SHOULD either be rejected, or sent verbatim when extending to the rendezvous point, in the same order received.

The service MAY reject the list of link specifiers if it is inconsistent with relay information from the directory, but SHOULD NOT modify it.

The ONION_KEY_TYPE field is:

[01] NTOR: ONION_KEY is 32 bytes long.

The ONION_KEY field describes the onion key that must be used when extending to the rendezvous point. It must be of a type listed as supported in the hidden service descriptor.

```
ClientNonce                                   [32 octets]
```

Where,

- ClientNonce is 32 octets of random data.

Then, the server replies with:

```
   ServerHash                                 [32 octets]
   ServerNonce                                [32 octets]

  Where,
  + ServerHash is computed as:
      HMAC-SHA256(CookieString,
        "ExtORPort authentication server-to-client hash" | ClientNonce
| ServerNonce)
  + ServerNonce is 32 random octets.
```

Upon receiving that data, the client computes ServerHash, and validates it against the ServerHash provided by the server.

If the server-provided ServerHash is invalid, the client MUST terminate the connection.

Otherwise the client replies with:

```
ClientHash                                    [32 octets]

  Where,
  + ClientHash is computed as:
      HMAC-SHA256(CookieString,
        "ExtORPort authentication client-to-server hash" | ClientNonce
| ServerNonce)
```

Upon receiving that data, the server computes ClientHash, and validates it against the ClientHash provided by the client.

Finally, the server replies with:

Status [1 octet]

```
  Where,
  + Status is 1 if the authentication was successful. If the
    authentication failed, Status is 0.
```

```
   + AuthType is the authentication scheme that the client wants to use
     for this session. A valid authentication type takes values from 1
to
     255. A value of 0 means that the client did not like the
     authentication types offered by the server.
```

If the client sent an AuthType of value 0, or an AuthType that the server does not support, the server MUST close the connection.

Authentication type: SAFE_COOKIE

We define one authentication type: SAFE_COOKIE. Its AuthType value is 1. It is based on the client proving to the bridge that it can access a given “cookie” file on disk. The purpose of authentication is to defend against cross-protocol attacks.

If the Extended ORPort is enabled, Tor should regenerate the cookie file on startup and store it in $DataDirectory/extended_orport_auth_cookie.

The location of the cookie can be overridden by using the configuration file parameter ExtORPortCookieAuthFile, which is defined as:

```
ExtORPortCookieAuthFile <path>
```

where `<path>` is a filesystem path.

Cookie-file format

The format of the cookie-file is:

```
      StaticHeader                                [32 octets]
      Cookie                                      [32 octets]

   Where,
   + StaticHeader is the following string:
     "! Extended ORPort Auth Cookie !\x0a"
   + Cookie is the shared-secret. During the SAFE_COOKIE protocol, the
     cookie is called CookieString.
```

Extended ORPort clients MUST make sure that the StaticHeader is present in the cookie file, before proceeding with the authentication protocol.

SAFE_COOKIE Protocol specification

A client that performs the SAFE_COOKIE handshake begins by sending:

The PAD field should be filled with zeros; its size should be chosen so that the INTRODUCE2 message occupies a fixed maximum size, in order to hide the length of the encrypted data. (This maximum size is 490, since we assume that a future Tor implementations will implement proposal 340 and thus lower the number of bytes that can be contained in a single relay message.) Note also that current versions of Tor only pad the INTRODUCE2 message up to 246 bytes.

Upon receiving a well-formed INTRODUCE2 message, the hidden service host will have:

```
   * The information needed to connect to the client's chosen
     rendezvous point.
   * The second half of a handshake to authenticate and establish a
     shared key with the hidden service client.
   * A set of shared keys to use for end-to-end encryption.
```

The same rules for multiplicity, ordering, and handling unknown types apply to the extension fields here as described [EST_INTRO] above.

More notes on replay resistance

Implementations SHOULD prevent INTRODUCE replays as well as reasonably practical. It is not, however, necessary to ensure that replays are completely impossible, so long as the opportunity for replay attacks remains limited. For example, it is not necessary to fsync() data to disk after each request.

> Rationale:
>
> The main reason we prevent INTRODUCE replays is to detect attempts by introduction points to mount replay attacks. Such attacks would cause the onion service to make a second circuit to the client’s chosen rendezvous point. If the attacker controls both the introduction point and the rendezvous point, they can use this to learn which original user circuit corresponded to the replayed request. This likely helps with traffic analysis somewhat, probably not *too* much if it can only be done occasionally.
>
> A second reason for preventing replays is to avoid DoS attacks: a single accepted INTRODUCE message causes an onion service to perform significant work, but doesn’t cost the sender too much in CPU or bandwidth.
>
> These attacks are *relatively* minor when only a limited number of messages can be successfully replayed, especially if the attacker can’t predict which replays will be detected. They become more problematic only when several messages can be replayed many times.

INTRODUCE1/INTRODUCE2 Extensions

The following sections details the currently supported or reserved extensions of an `INTRODUCE1`/`INTRODUCE2` message.

Note that there are two sets of extensions in `INTRODUCE1`/`INTRODUCE2`: one in the top-level, unencrypted portion, and one within the plaintext of ENCRYPTED (ie, after RENDEZVOUS_COOKIE and before ONION_KEY_TYPE.

The sets of extensions allowed in each part of the message are disjoint: each extension is valid in only *one* of the two places.

Nevertheless, for historical reasons, both kinds of extension are listed in this section, and they use nonoverlapping values of `EXT_FIELD_TYPE`.

Encrypted extensions are semantically equivalent to, and share a namespace with, the extensions used with circuit creation handshakes.

Congestion Control

This is used to request that the rendezvous circuit with the service be configured with congestion control.

EXT_FIELD_TYPE:

```
\[01\] -- CC_FIELD_REQUEST.
```

This extension has zero body length. Its presence signifies that the client wants to use congestion control. The client MUST NOT set this field if the service did not list "2" in the `FlowCtrl` line in the descriptor. The client SHOULD NOT provide this field if the consensus parameter 'cc_alg' is 0.

This extension is equivalent to the `CC_FIELD_REQUEST` extension used in circuit creation, except that no response is sent by the onion service.

This appears in the ENCRYPTED section of the INTRODUCE1/INTRODUCE2 message.

Proof-of-Work (PoW)

This extension can be used to optionally attach a proof of work to the introduction request. The proof must be calculated using unique parameters appropriate for this specific service. An acceptable proof will raise the priority of this introduction request according to the proof's verified computational effort.

Extended ORPort for pluggable transports

George Kadianakis, Nick Mathewson

Overview

This document describes the "Extended ORPort" protocol, a wrapper around Tor's ordinary ORPort protocol for use by bridges that support pluggable transports. It provides a way for server-side PTs and bridges to exchange additional information before beginning the actual OR connection.

See `tor-spec.txt` for information on the regular OR protocol, and `pt-spec.txt` for information on pluggable transports.

This protocol was originally proposed in proposal 196, and extended with authentication in proposal 217.

Establishing a connection and authenticating

When a client (that is to say, a server-side pluggable transport) connects to an Extended ORPort, the server sends:

```
     AuthTypes                                       [variable]
     EndAuthTypes                                    [1 octet]

   Where,

   + AuthTypes are the authentication schemes that the server supports
     for this session. They are multiple concatenated 1-octet values
 that
     take values from 1 to 255.
   + EndAuthTypes is the special value 0.
```

The client reads the list of supported authentication schemes, chooses one, and sends it back:

AuthType [1 octet]

Where,

References

```
    [REF_EQUIX]: https://github.com/tevador/equix
                 https://github.com/tevador/equix/blob/master/
devlog.md
    [REF_TABLE]: The table is based on the script below plus some
manual editing for readability:
                 https://gist.github.com/asn-
d6/99a936b0467b0cef88a677baaf0bbd04
    [REF_BOTNET]: https://media.kasperskycontenthub.com/wp-content/
uploads/sites/43/2009/07/01121538/ynam_botnets_0907_en.pdf
    [REF_CREDS]: https://lists.torproject.org/pipermail/tor-dev/2020-
March/014198.html
    [REF_TARGET]: https://en.bitcoin.it/wiki/Target
    [REF_TEVADOR_2]: https://lists.torproject.org/pipermail/tor-dev/
2020-June/014358.html
    [REF_TEVADOR_SIM]: https://github.com/mikeperry-tor/scratchpad/
blob/master/tor-pow/effort_sim.py#L57
```

This is for the proof-of-work DoS mitigation, described in depth by the Proof of Work for onion service introduction specification.

This appears in the ENCRYPTED section of the INTRODUCE1/INTRODUCE2 message.

The content is defined as follows:

EXT_FIELD_TYPE:

[02] – `PROOF_OF_WORK`

```
The EXT_FIELD content format is:

    POW_SCHEME     [1 byte]
    POW_NONCE      [16 bytes]
    POW_EFFORT     [4 bytes]
    POW_SEED       [4 bytes]
    POW_SOLUTION   [16 bytes]

where:

POW_SCHEME is 1 for the `v1` protocol specified here
POW_NONCE is the nonce value chosen by the client's solver
POW_EFFORT is the effort value chosen by the client,
           as a 32-bit integer in network byte order
POW_SEED identifies which seed was in use, by its first 4 bytes
POW_SOLUTION is a matching proof computed by the client's solver
```

Only SCHEME 1, `v1`, is currently defined. Other schemes may have a different format, after the POW_SCHEME byte. A correctly functioning client only submits solutions with a scheme and seed which were advertised by the server (using a “pow-params” Item in the HS descriptor) and have not yet expired. An extension with an unknown scheme or expired seed is suspicious and SHOULD result in introduction failure.

Introduced in tor-0.4.8.1-alpha.

Subprotocol Request

[RESERVED]

EXT_FIELD_TYPE:

```
  \[03\] -- Subprotocol Request
```

Introduction handshake encryption requirements

When decoding the encrypted information in an INTRODUCE2 message, a hidden service host must be able to:

```
     * Decrypt additional information included in the INTRODUCE2
message,
       to include the rendezvous token and the information needed to
       extend to the rendezvous point.

     * Establish a set of shared keys for use with the client.

     * Authenticate that the message has not been modified since the
client
       generated it.
```

Note that the old TAP-derived protocol of the previous hidden service design achieved the first two requirements, but not the third.

Encryption handshake: hs-ntor

```
TODO: relocate this
```

This is a variant of the ntor handshake (also see "Anonymity and one-way authentication in key-exchange protocols" by Goldberg, Stebila, and Ustaoglu).

It behaves the same as the ntor handshake, except that, in addition to negotiating forward secure keys, it also provides a means for encrypting non-forward-secure "additional data" to the server (in this case, to the hidden service host) as part of the handshake. This "additional data" is used to encode encrypted introduce1 extensions.

Notation here is as for the in section 5.1.4 of tor-spec.txt, which defines the ntor handshake.

The PROTOID for this variant is "tor-hs-ntor-curve25519-sha3-256-1". We also use the following tweak values:

```
       t_hsenc    = PROTOID | ":hs_key_extract"
       t_hsverify = PROTOID | ":hs_verify"
       t_hsmac    = PROTOID | ":hs_mac"
       m_hsexpand = PROTOID | ":hs_key_expand"
```

3. Connection data read (top half)

   Now that we have the above pieces, we can use them to measure just the "top half" part of the procedure. That's when bytes are taken from the connection inbound buffer and parsed into an INTRODUCE2 message where basic validation is done.

   There is an average of 2.42 INTRODUCE2 messages per mainloop event and so we divide that by the full mainloop event mean time to get the time for one message. From that we subtract the "bottom half" mean time to get how much the "top half" takes:

```
=> 13.43 / (7931 / 3279) = 5.55
=> 5.55 - 5.29 = 0.26

Mean: 0.26 msec
```

To summarize, during our measurements the average number of INTRODUCE2 messages a mainloop event processed is ~2.42 messages (7931 messages for 3279 mainloop invocations).

This means that, taking the mean of mainloop event times, it takes ~5.55msec (13.43/2.42) to completely process an INTRODUCE2 messages. Then if we look deeper we see that the "top half" of INTRODUCE2 message processing takes 0.26 msec in average, whereas the "bottom half" takes around 5.33 msec.

The heavyness of the "bottom half" is to be expected since that's where 95% of the total work takes place: in particular the rendezvous path selection and circuit launch.

performance measurements we might learn that it's preferable to bump the number of requests in the future from 1 to N where N <= 32.

Performance measurements

This section will detail the performance measurements we've done on `tor.git` for handling an INTRODUCE2 message and then a discussion on how much more CPU time we can add (for PoW validation) before it badly degrades our performance.

Tor measurements

In this section we will derive measurement numbers for the "top half" and "bottom half" parts of handling an introduction request.

These measurements have been done on tor.git at commit `80031db32abebaf4d0a91c01db258fcdbd54a471`.

We've measured several set of actions of the INTRODUCE2 message handling process on Intel(R) Xeon(R) CPU E5-2650 v4. Our service was accessed by an array of clients that sent introduction requests for a period of 60 seconds.

1. Full Mainloop Event

   We start by measuring the full time it takes for a mainloop event to process an inbuf containing INTRODUCE2 messages. The mainloop event processed 2.42 messages per invocation on average during our measurements.

   ```
   Total measurements: 3279

     Min: 0.30 msec - 1st Q.: 5.47 msec - Median: 5.91 msec
     Mean: 13.43 msec - 3rd Q.: 16.20 msec - Max: 257.95 msec
   ```

2. INTRODUCE2 message processing (bottom-half)

   We also measured how much time the "bottom half" part of the process takes. That's the heavy part of processing an introduction request as seen in step (4) of the main loop section above:

   ```
   Total measurements: 7931

     Min: 0.28 msec - 1st Q.: 5.06 msec - Median: 5.33 msec
     Mean: 5.29 msec - 3rd Q.: 5.57 msec - Max: 14.64 msec
   ```

To make an INTRODUCE1 message, the client must know a public encryption key B for the hidden service on this introduction circuit. The client generates a single-use keypair:

x,X = KEYGEN()

and computes:

```
             intro_secret_hs_input = EXP(B,x) | AUTH_KEY | X | B |
PROTOID
             info = m_hsexpand | N_hs_subcred
             hs_keys = SHAKE256_KDF(intro_secret_hs_input | t_hsenc |
info, S_KEY_LEN+MAC_LEN)
             ENC_KEY = hs_keys[0:S_KEY_LEN]
             MAC_KEY = hs_keys[S_KEY_LEN:S_KEY_LEN+MAC_KEY_LEN]

   and sends, as the ENCRYPTED part of the INTRODUCE1 message:

         CLIENT_PK               [PK_PUBKEY_LEN bytes]
         ENCRYPTED_DATA          [Padded to length of plaintext]
         MAC                     [MAC_LEN bytes]
```

Substituting those fields into the INTRODUCE1 message body format described in [FMT_INTRO1] above, we have

```
            LEGACY_KEY_ID               [20 bytes]
            AUTH_KEY_TYPE               [1 byte]
            AUTH_KEY_LEN                [2 bytes]
            AUTH_KEY                    [AUTH_KEY_LEN bytes]
            N_EXTENSIONS                [1 bytes]
            N_EXTENSIONS times:
               EXT_FIELD_TYPE           [1 byte]
               EXT_FIELD_LEN            [1 byte]
               EXT_FIELD                [EXT_FIELD_LEN bytes]
            ENCRYPTED:
               CLIENT_PK                [PK_PUBKEY_LEN bytes]
               ENCRYPTED_DATA           [Padded to length of
plaintext]
               MAC                      [MAC_LEN bytes]
```

(This format is as documented in [FMT_INTRO1] above, except that here we describe how to build the ENCRYPTED portion.)

Here, the encryption key plays the role of B in the regular ntor handshake, and the AUTH_KEY field plays the role of the node ID. The CLIENT_PK field is the public key X. The ENCRYPTED_DATA field is the message plaintext, encrypted with the symmetric key ENC_KEY. The MAC field is a MAC of all of the message from the

AUTH_KEY through the end of ENCRYPTED_DATA, using the MAC_KEY value as its key.

To process this format, the hidden service checks PK_VALID(CLIENT_PK) as necessary, and then computes ENC_KEY and MAC_KEY as the client did above, except using EXP(CLIENT_PK,b) in the calculation of intro_secret_hs_input. The service host then checks whether the MAC is correct. If it is invalid, it drops the message. Otherwise, it computes the plaintext by decrypting ENCRYPTED_DATA.

The hidden service host now completes the service side of the extended ntor handshake, as described in tor-spec.txt section 5.1.4, with the modified PROTOID as given above. To be explicit, the hidden service host generates a keypair of y,Y = KEYGEN(), and uses its introduction point encryption key 'b' to compute:

```
      intro_secret_hs_input = EXP(X,b) | AUTH_KEY | X | B | PROTOID
      info = m_hsexpand | N_hs_subcred
      hs_keys = SHAKE256_KDF(intro_secret_hs_input | t_hsenc | info,
S_KEY_LEN+MAC_LEN)
      HS_DEC_KEY = hs_keys[0:S_KEY_LEN]
      HS_MAC_KEY = hs_keys[S_KEY_LEN:S_KEY_LEN+MAC_KEY_LEN]

      (The above are used to check the MAC and then decrypt the
      encrypted data.)

      rend_secret_hs_input = EXP(X,y) | EXP(X,b) | AUTH_KEY | B | X |
Y | PROTOID
      NTOR_KEY_SEED = MAC(rend_secret_hs_input, t_hsenc)
      verify = MAC(rend_secret_hs_input, t_hsverify)
      auth_input = verify | AUTH_KEY | B | Y | X | PROTOID | "Server"
      AUTH_INPUT_MAC = MAC(auth_input, t_hsmac)

      (The above are used to finish the ntor handshake.)

   The server's handshake reply is:

       SERVER_PK   Y                         [PK_PUBKEY_LEN bytes]
       AUTH        AUTH_INPUT_MAC            [MAC_LEN bytes]
```

These fields will be sent to the client in a RENDEZVOUS1 message using the HANDSHAKE_INFO element (see [JOIN_REND]).

The hidden service host now also knows the keys generated by the handshake, which it will use to encrypt and authenticate data end-to-end between the client and the server. These keys are as computed with the ntor handshake, except that instead of using AES-128 and SHA1 for this hop, we use AES-256 and SHA3-256.

Top half and bottom half

The top half process is responsible for queuing introductions into the priority queue as follows:

1. Unpack cell from inbuf to local buffer.
2. Decrypt cell (AES operations).
3. Parse INTRODUCE2 message and validate PoW.
4. Return to mainloop event which essentially means step (1).

The top-half basically does all operations from the main loop section above, excepting (4).

An then, the bottom-half process is responsible for handling introductions and doing rendezvous. To achieve this we introduce a new mainloop event to process the priority queue *after* the top-half event has completed. This new event would do these operations sequentially:

1. Pop INTRODUCE2 message from priority queue.
2. Parse and process INTRODUCE2 message.
3. End event and yield back to mainloop.

Scheduling the bottom half process

The question now becomes: when should the "bottom half" event get triggered from the mainloop?

We propose that this event is scheduled in when the network I/O event queues at least 1 request into the priority queue. Then, as long as it has a request in the queue, it would re-schedule itself for immediate execution meaning at the next mainloop round, it would execute again.

The idea is to try to empty the queue as fast as it can in order to provide a fast response time to an introduction request but always leave a chance for more requests to appear between request processing by yielding back to the mainloop. With this we are aiming to always have the most up-to-date version of the priority queue when we are completing introductions: this way we are prioritizing clients that spent a lot of time and effort completing their PoW.

If the size of the queue drops to 0, it stops scheduling itself in order to not create a busy loop. The network I/O event will re-schedule it in time.

Notice that the proposed solution will make the service handle 1 single introduction request at every main loop event. However, when we do

3. Parse relay message and process it depending on its RELAY_COMMAND.
4. INTRODUCE2 handling which means building a rendezvous circuit:
    - Path selection
    - Launch circuit to first hop.
5. Return to mainloop event which essentially means back to step (1).

Tor will read at most 32 messages out of the inbuf per mainloop round.

Requirements for PoW

With this proposal, in order to prioritize requests by the amount of PoW work it has done, requests can *not* be processed sequentially as described above.

Thus, we need a way to queue a certain number of requests, prioritize them and then process some request(s) from the top of the queue (that is, the requests that have done the most PoW effort).

We thus require a new request processing flow that is *not* compatible with current tor design. The elements are:

- Validate PoW and place requests in a priority queue of introduction requests (the introduction queue).
- Defer "bottom half" request processing for after requests have been queued into the priority queue.

Proposed scheduler

The intuitive way to address the queueing requirements above would be to do this simple and naive approach:

1. Mainloop: Empty inbuf of introduction requests into priority queue
2. Process all requests in pqueue
3. Goto (1)

However, we are worried that handling all those requests before returning to the mainloop opens possibilities of attack by an adversary since the priority queue is not gonna be kept up to date while we process all those requests. This means that we might spend lots of time dealing with introductions that don't deserve it.

We thus propose to split the INTRODUCE2 handling into two different steps: "top half" and "bottom half" process.

Authentication during the introduction phase.

Hidden services may restrict access only to authorized users. One mechanism to do so is the credential mechanism, where only users who know the credential for a hidden service may connect at all.

There is one defined authentication type: `ed25519`.

Ed25519-based authentication `ed25519`

(NOTE: This section is not implemented by Tor. It is likely that we would want to change its design substantially before deploying any implementation. At the very least, we would want to bind these extensions to a single onion service, to prevent replays. We might also want to look for ways to limit the number of keys a user needs to have.)

To authenticate with an Ed25519 private key, the user must include an extension field in the encrypted part of the INTRODUCE1 message with an EXT_FIELD_TYPE type of [TBD] and the contents:

```
Nonce     [16 bytes]
Pubkey    [32 bytes]
Signature [64 bytes]
```

Nonce is a random value. Pubkey is the public key that will be used to authenticate. [TODO: should this be an identifier for the public key instead?] Signature is the signature, using Ed25519, of:

```
"hidserv-userauth-ed25519"
Nonce       (same as above)
Pubkey      (same as above)
AUTH_KEY    (As in the INTRODUCE1 message)
```

The hidden service host checks this by seeing whether it recognizes and would accept a signature from the provided public key. If it would, then it checks whether the signature is correct. If it is, then the correct user has authenticated.

Replay prevention on the whole message is sufficient to prevent replays on the authentication.

Users SHOULD NOT use the same public key with multiple hidden services.

The rendezvous protocol

Before connecting to a hidden service, the client first builds a circuit to an arbitrarily chosen Tor node (known as the rendezvous point), and sends an ESTABLISH_RENDEZVOUS message. The hidden service later connects to the same node and sends a RENDEZVOUS message. Once this has occurred, the relay forwards the contents of the RENDEZVOUS message to the client, and joins the two circuits together.

Single Onion Services attempt to build a non-anonymous single-hop circuit, but use an anonymous 3-hop circuit if:

```
     * the rend point is on an address that is configured as
unreachable via
       a direct connection, or
     * the initial attempt to connect to the rend point over a single-
hop
       circuit fails, and they are retrying the rend point connection.
```

Establishing a rendezvous point

The client sends the rendezvous point a RELAY_COMMAND_ESTABLISH_RENDEZVOUS message containing a 20-byte value.

RENDEZVOUS_COOKIE [20 bytes]

Rendezvous points MUST ignore any extra bytes in an ESTABLISH_RENDEZVOUS message. (Older versions of Tor did not.)

The rendezvous cookie is an arbitrary 20-byte value, chosen randomly by the client. The client SHOULD choose a new rendezvous cookie for each new connection attempt. If the rendezvous cookie is already in use on an existing circuit, the rendezvous point should reject it and destroy the circuit.

Upon receiving an ESTABLISH_RENDEZVOUS message, the rendezvous point associates the cookie with the circuit on which it was sent. It replies to the client with an empty RENDEZVOUS_ESTABLISHED message to indicate success. Clients MUST ignore any extra bytes in a RENDEZVOUS_ESTABLISHED message.

The client MUST NOT use the circuit which sent the message for any purpose other than rendezvous with the given location-hidden service.

position to cause downstream resource consumption of nearly every type. Each relay involved experiences increased CPU load from the circuit floods they process. We think that asking legitimate clients to carry out PoW computations doesn't affect the equation too much, since an attacker right now can very quickly use the same resources that hundreds of legitimate clients do in a whole day.

We hope to make things better: The hope is that systems like this will make the DoS actors go away and hence the PoW system will not be used. As long as DoS is happening there will be a waste of energy, but if we manage to demotivate them with technical means, the network as a whole will less wasteful. Also see The DoS Catch-22.

Acknowledgements

Thanks a lot to tevador for the various improvements to the proposal and for helping us understand and tweak the RandomX scheme.

Thanks to Solar Designer for the help in understanding the current PoW landscape, the various approaches we could take, and teaching us a few neat tricks.

Scheduler implementation for C tor

This section describes how we will implement this proposal in the "tor" software (little-t tor).

The following should be read as if tor is an onion service and thus the end point of all inbound data.

The Main Loop

Tor uses libevent for its mainloop. For network I/O operations, a mainloop event is used to inform tor if it can read on a certain socket, or a connection object in tor.

From there, this event will empty the connection input buffer (inbuf) by extracting and processing a request at a time. The mainloop is single threaded and thus each request is handled sequentially.

Processing an INTRODUCE2 message at the onion service means a series of operations (in order):

1. Unpack relay cell from inbuf to local buffer.
2. Decrypt cell (AES operations).

Future designs

This is just the beginning in DoS defences for Tor and there are various future designs and schemes that we can investigate. Here is a brief summary of these:

- "More advanced PoW schemes" – We could use more advanced memory-hard PoW schemes like MTP-argon2 or Itsuku to make it even harder for adversaries to create successful PoWs. Unfortunately these schemes have much bigger proof sizes, and they won't fit in INTRODUCE1 messages. See #31223 for more details.
- "Third-party anonymous credentials" – We can use anonymous credentials and a third-party token issuance server on the clearnet to issue tokens based on PoW or CAPTCHA and then use those tokens to get access to the service. See `[REF_CREDS]` for more details.
- "PoW + Anonymous Credentials" – We can make a hybrid of the above ideas where we present a hard puzzle to the user when connecting to the onion service, and if they solve it we then give the user a bunch of anonymous tokens that can be used in the future. This can all happen between the client and the service without a need for a third party.

All of the above approaches are much more complicated than the `v1` design, and hence we want to start easy before we get into more serious projects. The current implementation requires complexity within the Equi-X implementation but its impact on the overall tor network can be relatively simple.

Environment

This algorithm shares a broad concept, proof of work, with some notoriously power hungry and wasteful software. We love the environment, and we too are concerned with how proof of work schemes typically waste huge amounts of energy by doing useless hash iterations.

Nevertheless, there are some massive differences in both the scale and the dynamics of what we are doing here: we are performing fairly small amounts of computation, and it's used as part of a scheme to disincentivize attacks entirely. If we do our job well, people stop computing these proof-of-work functions entirely and find something else to attack.

We think we aren't making a bad situation worse: DoS attacks on the Tor network are already happening and attackers are already burning energy to carry them out. As we see in the denial-of-service overview, attacks on onion services are in a

The client should establish a rendezvous point BEFORE trying to connect to a hidden service.

Joining to a rendezvous point

To complete a rendezvous, the hidden service host builds a circuit to the rendezvous point and sends a RENDEZVOUS1 message containing:

```
RENDEZVOUS_COOKIE          [20 bytes]
HANDSHAKE_INFO             [variable; depends on handshake type
                            used.]
```

where RENDEZVOUS_COOKIE is the cookie suggested by the client during the introduction (see [PROCESS_INTRO2]) and HANDSHAKE_INFO is defined in [NTOR-WITH-EXTRA-DATA].

If the cookie matches the rendezvous cookie set on any not-yet-connected circuit on the rendezvous point, the rendezvous point connects the two circuits, and sends a RENDEZVOUS2 message to the client containing the HANDSHAKE_INFO field of the RENDEZVOUS1 message.

Upon receiving the RENDEZVOUS2 message, the client verifies that HANDSHAKE_INFO correctly completes a handshake. To do so, the client parses SERVER_PK from HANDSHAKE_INFO and reverses the final operations of section [NTOR-WITH-EXTRA-DATA] as shown here:

```
      rend_secret_hs_input = EXP(Y,x) | EXP(B,x) | AUTH_KEY | B | X |
Y | PROTOID
      NTOR_KEY_SEED = MAC(rend_secret_hs_input, t_hsenc)
      verify = MAC(rend_secret_hs_input, t_hsverify)
      auth_input = verify | AUTH_KEY | B | Y | X | PROTOID | "Server"
      AUTH_INPUT_MAC = MAC(auth_input, t_hsmac)
```

Finally the client verifies that the received AUTH field of HANDSHAKE_INFO is equal to the computed AUTH_INPUT_MAC.

Now both parties use the handshake output to derive shared keys for use on the circuit as specified in the section below:

Key expansion

The hidden service and its client need to derive crypto keys from the `NTOR_KEY_SEED` part of the handshake output. To do so, they use the KDF construction as follows:

```
K = KDF(NTOR_KEY_SEED | m_hsexpand,    SHA3_256_LEN *2 + S_KEY_LEN* 2)
```

The first SHA3_256_LEN bytes of K form the forward digest Df; the next SHA3_256_LEN bytes form the backward digest Db; the next S_KEY_LEN bytes form Kf, and the final S_KEY_LEN bytes form Kb. Excess bytes from K are discarded.

Subsequently, the rendezvous point passes RELAY cells, unchanged, from each of the two circuits to the other. When Alice's OP sends RELAY cells along the circuit, it authenticates with Df, and encrypts them with the Kf, then with all of the keys for the ORs in Alice's side of the circuit; and when Alice's OP receives RELAY cells from the circuit, it decrypts them with the keys for the ORs in Alice's side of the circuit, then decrypts them with Kb, and checks integrity with Db. Bob's OP does the same, with Kf and Kb interchanged.

[TODO: Should we encrypt HANDSHAKE_INFO as we did INTRODUCE2 contents? It's not necessary, but it could be wise. Similarly, we should make it extensible.]

Using legacy hosts as rendezvous points

[This section is obsolete and refers to a workaround for now-obsolete Tor relay versions. It is included for historical reasons.]

The behavior of ESTABLISH_RENDEZVOUS is unchanged from older versions of this protocol, except that relays should now ignore unexpected bytes at the end.

Old versions of Tor required that RENDEZVOUS message bodies be exactly 168 bytes long. All shorter rendezvous bodies should be padded to this length with random bytes, to make them difficult to distinguish from older protocols at the rendezvous point.

Relays older than 0.2.9.1 should not be used for rendezvous points by next generation onion services because they enforce too-strict length checks to RENDEZVOUS messages. Hence the "HSRend" protocol from proposal#264 should be used to select relays for rendezvous points.

- For a PoW function with a 10ms verification time, an attacker needs to send 64 high-effort introduction requests per second to succeed in a bottom-half attack attack.

We can use this table to specify a starting difficulty that won't allow our target adversary to succeed in an bottom-half attack attack.

Note that in practice verification times are much lower; the scale of the above table does not match the current implementation's reality.

User experience

This proposal has user facing UX consequences.

When the client first attempts a pow, it can note how long iterations of the hash function take, and then use this to determine an estimation of the duration of the PoW. This estimation could be communicated via the control port or other mechanism, such that the browser could display how long the PoW is expected to take on their device. If the device is a mobile platform, and this time estimation is large, it could recommend that the user try from a desktop machine.

Future work

Incremental improvements to this proposal

There are various improvements that can be done in this proposal, and while we are trying to keep this `v1` version simple, we need to keep the design extensible so that we build more features into it. In particular:

- End-to-end introduction ACKs

  This proposal suffers from various UX issues because there is no end-to-end mechanism for an onion service to inform the client about its introduction request. If we had end-to-end introduction ACKs many of the problems seen in client-side effort estimation would be alleviated. The problem here is that end-to-end ACKs require modifications on the introduction point code and a network update which is a lengthy process.

- Multithreading scheduler

  Our scheduler is pretty limited by the fact that Tor has a single-threaded design. If we improve our multithreading support we could handle a much greater amount of introduction requests per second.

- If an adversary has access to 50 boxes, and we want to limit her to 5 successes per second, then a legitimate client with a single box should be expected to spend 10 seconds getting a single success.
- If an adversary has access to 5 boxes, and we want to limit her to 5 successes per second, then a legitimate client with a single box should be expected to spend 1 seconds getting a single success.

With the above table we can create some profiles for starting values of our PoW difficulty.

Analysis based on Tor's performance

To go deeper here, we can use the performance measurements on tor to get a more specific intuition on the starting difficulty. In particular, we learned that completely handling an introduction request takes 5.55 msecs in average. Using that value, we can compute the following table, that describes the number of introduction requests we can handle per second for different values of PoW verification:

PoW Verification Time	Total time to handle request	Requests handled per second
0 msec	5.55 msec	180.18
1 msec	6.55 msec	152.67
2 msec	7.55 msec	132.45
3 msec	8.55 msec	116.96
4 msec	9.55 mesc	104.71
5 msec	10.55 msec	94.79
6 msec	11.55 msec	86.58
7 msec	12.55 msec	79.68
8 msec	13.55 msec	73.80
9 msec	14.55 msec	68.73
10 msec	15.55 msec	64.31

Here is how you can read the table above:

- For a PoW function with a 1ms verification time, an attacker needs to send 152 high-effort introduction requests per second to succeed in a bottom-half attack attack.

Encrypting data between client and host

A successfully completed handshake, as embedded in the INTRODUCE/ RENDEZVOUS messages, gives the client and hidden service host a shared set of keys Kf, Kb, Df, Db, which they use for sending end-to-end traffic encryption and authentication as in the regular Tor relay encryption protocol, applying encryption with these keys before other encryption, and decrypting with these keys before other decryption. The client encrypts with Kf and decrypts with Kb; the service host does the opposite.

As mentioned previously, these keys are used the same as for regular relay cell encryption, except that instead of using AES-128 and SHA1, both parties use AES-256 and SHA3-256.

Encoding onion addresses [ONIONADDRESS]

The onion address of a hidden service includes its identity public key, a version field and a basic checksum. All this information is then base32 encoded as shown below:

```
      onion_address = base32(PUBKEY | CHECKSUM | VERSION) + ".onion"
      CHECKSUM = SHA3_256(".onion checksum" | PUBKEY | VERSION)[:2]

      where:
        - PUBKEY is the 32 bytes ed25519 master pubkey (KP_hs_id) of
the hidden service.
        - VERSION is a one byte version field (default value '\x03')
        - ".onion checksum" is a constant string
        - CHECKSUM is truncated to two bytes before inserting it in
onion_address

  Here are a few example addresses:

        pg6mmjiyjmcrsslvykfwnntlaru7p5svn6y2ymmju6nubxndf4pscryd.onion
        sp3k262uwy4r2k3ycr5awluarykdpag6a7y33jxop4cs2lu5uz5sseqd.onion
        xa4r2iadxm55fbnqgwwi5mymqdcofiu3w6rpbtqn7b2dyn7mgwj64jyd.onion
```

For historical notes and rationales about this encoding, see this discussion thread.

```
       ================================================================
       |    Expected Time (in seconds) Per Success For One Machine   |

=================================================================================
=====

|
    |
 |   Attacker Succeses        1       5       10      20      30
50    |
 |       per
second                                                          |

|
    |
 |            5               5       1       0       0       0
0     |
 |            50              50      10      5       2       1
1     |
 |            100             100     20      10      5       3
2     |
 | Attacker   200             200     40      20      10      6
4     |
 |  Boxes     300             300     60      30      15      10
6     |
 |            400             400     80      40      20      13
8     |
 |            500             500     100     50      25      16
10    |
 |            1000            1000    200     100     50      33
20    |

|
    |

=================================================================================
======
```

Here is how you can read the table above:

- If an adversary has a botnet with 1000 boxes, and we want to limit her to 1 success per second, then a legitimate client with a single box should be expected to spend 1000 seconds getting a single success.
- If an adversary has a botnet with 1000 boxes, and we want to limit her to 5 successes per second, then a legitimate client with a single box should be expected to spend 200 seconds getting a single success.
- If an adversary has a botnet with 500 boxes, and we want to limit her to 5 successes per second, then a legitimate client with a single box should be expected to spend 100 seconds getting a single success.

PoW difficulty analysis

The difficulty setting of our PoW basically dictates how difficult it should be to get a success in our PoW system. An attacker who can get many successes per second can pull a successful bottom-half attack against our system.

In classic PoW systems, "success" is defined as getting a hash output below the "target". However, since our system is dynamic, we define "success" as an abstract high-effort computation.

The original analysis here concluded that we still need a starting difficulty setting that will be used for bootstrapping the system. The client and attacker can still aim higher or lower but for UX purposes and for analysis purposes it was useful to define a starting difficulty, to minimize retries by clients.

In current use it was found that an effort of 1 makes a fine minimum, so we don't normally have a concept of minimum effort. Consider the actual "minimum effort" in `v1` now to simply be the expected runtime of one single Equi-X solve.

Analysis based on adversary power

In this section we will try to do an analysis of PoW difficulty without using any sort of Tor-related or PoW-related benchmark numbers.

We created the table (see `[REF_TABLE]`) below which shows how much time a legitimate client with a single machine should expect to burn before they get a single success.

The x-axis is how many successes we want the attacker to be able to do per second: the more successes we allow the adversary, the more they can overwhelm our introduction queue. The y-axis is how many machines the adversary has in her disposal, ranging from just 5 to 1000.

Managing streams

Sending BEGIN messages

In order to open a new stream to an onion service, the client sends a BEGIN message on an established rendezvous circuit.

When sending a BEGIN message to an onion service, a client should use an empty string as the target address, and not set any flags on the begin message.

> For example, to open a connection to `<some_addr>.onion` on port 443, a client would send a BEGIN message with the address:port string of `":443"`, and a `FLAGS` value of 0. The 0-values `FLAGS` would not be encoded, according to the instructions for encoding BEGIN messages.

Receiving BEGIN messages

When a service receives a BEGIN message, it should check its port, *and ignore all other fields in the begin message*, including its address and flags.

If a service chooses to reject a BEGIN message, it should typically destroy the circuit entirely to prevent port scanning, resource exhaustion, and other undesirable behaviors. But if it rejects the BEGIN without destroy the circuit, it should send back an `END` message with the `DONE` reason, to avoid leaking any further information.

If the service chooses to accept the BEGIN message, it should send back a CONNECTED message with an empty body.

References

How can we improve the HSDir unpredictability design proposed in [SHAREDRANDOM]? See these references for discussion.

```
[SHAREDRANDOM-REFS]:
        https://gitweb.torproject.org/torspec.git/tree/proposals/250-
commit-reveal-consensus.txt
        https://trac.torproject.org/projects/tor/ticket/8244
```

Scaling hidden services is hard. There are on-going discussions that you might be able to help with:

```
[SCALING-REFS]:
        https://lists.torproject.org/pipermail/tor-dev/2013-October/
005556.html
```

How can hidden service addresses become memorable while retaining their self-authenticating and decentralized nature? See these references for some proposals; many more are possible.

```
[HUMANE-HSADDRESSES-REFS]:
        https://gitweb.torproject.org/torspec.git/blob/HEAD:/
proposals/ideas/xxx-onion-nyms.txt
        http://archives.seul.org/or/dev/Dec-2011/msg00034.html
```

Hidden Services are pretty slow. Both because of the lengthy setup procedure and because the final circuit has 6 hops. How can we make the Hidden Service protocol faster? See these references for some suggestions.

```
[PERFORMANCE-REFS]:
        "Improving Efficiency and Simplicity of Tor circuit
        establishment and hidden services" by Overlier, L., and
        P. Syverson

        [TODO: Need more here! Do we have any? :( ]
```

Other references:

PoW Verification Time	Total “top half” time	Requests Queued per second
1 msec	1.26 msec	793
2 msec	2.26 msec	442
3 msec	3.26 msec	306
4 msec	4.26 msec	234
5 msec	5.26 msec	190
6 msec	6.26 msec	159
7 msec	7.26 msec	137
8 msec	8.26 msec	121
9 msec	9.26 msec	107
10 msec	10.26 msec	97

Here is how you can read the table above:

- For a PoW function with a 1ms verification time, an attacker needs to send 793 dummy introduction requests per second to succeed in a top-half attack.
- For a PoW function with a 2ms verification time, an attacker needs to send 442 dummy requests per second to succeed in a top-half attack.
- For a PoW function with a 10ms verification time, an attacker needs to send 97 dummy requests per second to succeed in a top-half attack.

Whether an attacker can succeed at that depends on the attacker’s resources, but also on the network’s capacity.

Our purpose here is to have the smallest PoW verification overhead possible that also allows us to achieve all our other goals.

Note that the table above is simply the result of a naive multiplication and does not take into account all the auxiliary overheads that happen every second like the time to invoke the mainloop, the bottom-half processes, or pretty much anything other than the “top-half” processing.

During our measurements the time to handle introduction requests dominates any other action time: There might be events that require a long processing time, but these are pretty infrequent (like uploading a new HS descriptor) and hence over a long time they smooth out. Hence extrapolating the total introduction requests queued per second based on a single “top half” time seems like good enough to get some initial intuition. That said, the values of “Requests queued per second” from the table above, are likely much smaller than displayed above because of all the auxiliary overheads.

We first start by tuning the time it takes to verify a PoW token. We do this first because it's fundamental to the performance of onion services and can turn into a DoS vector of its own. We will do this tuning in a way that's agnostic to the chosen PoW function.

We previously considered the concept of a nonzero starting difficulty setting. This analysis still references such a concept, even though the currently recommended implementation uses a starting effort of zero. (We now expect early increases in effort during an attack to be driven primarily by client retry behavior.)

At the end of this section we will estimate the resources that an attacker needs to overwhelm the onion service, the resources that the service needs to verify introduction requests, and the resources that legitimate clients need to get to the onion service.

PoW verification

Verifying a PoW token is the first thing that a service does when it receives an INTRODUCE2 message. Our current implementation is described by the `v1` verification algorithm specification.

Verification time is a critical performance parameter. Actual times can be measured by `cargo bench` now, and the verification speeds we achieve are more like 50-120 microseconds. The specific numbers below are dated, but the analysys below is preserved as an illustration of the design space we are optimizing within.

To defend against a top-half attack it's important that we can quickly perform all the steps in-between receiving an introduction request over the network and adding it to our effort-prioritized queue.

All time spent verifying PoW adds overhead to the already existing "top half" part of handling an INTRODUCE message. Hence we should be careful to add minimal overhead here.

During our performance measurements on tor we learned that the "top half" takes about 0.26 msecs in average, without doing any sort of PoW verification. Using that value we compute the following table, that describes the number of requests we can queue per second (aka times we can perform the "top half" process) for different values of PoW verification time:

PoW Verification Time	Total "top half" time	Requests Queued per second
0 msec	0.26 msec	3846

```
[KEYBLIND-REFS]:
        https://trac.torproject.org/projects/tor/ticket/8106
        https://lists.torproject.org/pipermail/tor-dev/2012-September/
004026.html

[KEYBLIND-PROOF]:
        https://lists.torproject.org/pipermail/tor-dev/2013-December/
005943.html

[ATTACK-REFS]:
        "Trawling for Tor Hidden Services: Detection, Measurement,
        Deanonymization" by Alex Biryukov, Ivan Pustogarov,
        Ralf-Philipp Weinmann

        "Locating Hidden Servers" by Lasse Øverlier and Paul
        Syverson

[ED25519-REFS]:
        "High-speed high-security signatures" by Daniel
        J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and
        Bo-Yin Yang. http://cr.yp.to/papers.html#ed25519

[ED25519-B-REF]:
        https://tools.ietf.org/html/draft-josefsson-eddsa-
ed25519-03#section-5:

[SRV-TP-REFS]:
        https://lists.torproject.org/pipermail/tor-dev/2016-April/
010759.html

[VANITY-REFS]:
        https://github.com/Yawning/horse25519

[ONIONADDRESS-REFS]:
        https://lists.torproject.org/pipermail/tor-dev/2017-January/
011816.html

[TORSION-REFS]:
        https://lists.torproject.org/pipermail/tor-dev/2017-April/
012164.html
        https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-
cryptonote-based-currencies.html
```

Appendix A: Signature scheme with key blinding

Key derivation overview

As described in [IMD:DIST] and [SUBCRED] above, we require a "key blinding" system that works (roughly) as follows:

There is a master keypair (sk, pk).

```
        Given the keypair and a nonce n, there is a derivation
function
        that gives a new blinded keypair (sk_n, pk_n).  This keypair
can
        be used for signing.

        Given only the public key and the nonce, there is a function
        that gives pk_n.

        Without knowing pk, it is not possible to derive pk_n; without
        knowing sk, it is not possible to derive sk_n.

        It's possible to check that a signature was made with sk_n
while
        knowing only pk_n.

        Someone who sees a large number of blinded public keys and
        signatures made using those public keys can't tell which
        signatures and which blinded keys were derived from the same
        master keypair.

        You can't forge signatures.

        [TODO: Insert a more rigorous definition and better
references.]
```

Tor's key derivation scheme

We propose the following scheme for key blinding, based on Ed25519.

Hybrid overwhelm strategy

If both the top- and bottom- halves are processed by the same thread, this opens up the possibility for a "hybrid" attack. Given the performance figures for the bottom half (0.31 ms/req.) and the top half (5.5 ms/req.), the attacker can optimally deny service by submitting 91 high-effort requests and 1520 invalid requests per second. This will completely saturate the main loop because:

```
0.31*(1520+91) ~ 0.5 sec.
5.5*91         ~ 0.5 sec.
```

This attack only has half the bandwidth requirement of a top-half attack and half the compute requirement of a bottom-half attack..

Alternatively, the attacker can adjust the ratio between invalid and high-effort requests depending on their bandwidth and compute capabilities.

Gaming the effort control logic

Another way to beat this system is for the attacker to game the effort control logic. Essentially, there are two attacks that we are trying to avoid:

- Attacker sets descriptor suggested-effort to a very high value effectively making it impossible for most clients to produce a PoW token in a reasonable timeframe.
- Attacker sets descriptor suggested-effort to a very small value so that most clients aim for a small value while the attacker comfortably launches an bottom-half attack using medium effort PoW (see this post by tevador on tor-dev from May 2020).

Precomputed PoW attack

The attacker may precompute many valid PoW nonces and submit them all at once before the current seed expires, overwhelming the service temporarily even using a single computer. The current scheme gives the attackers 4 hours to launch this attack since each seed lasts 2 hours and the service caches two seeds.

An attacker with this attack might be aiming to DoS the service for a limited amount of time, or to cause an effort control attack.

Parameter tuning

There are various parameters in this PoW system that need to be tuned:

Analysis and discussion

Warning: Take all the PoW performance numbers on this page with a large grain of salt. Most of this is based on very early analysis that has not been updated for the current state of implementation.

For current performance numbers on a specific piece of hardware, please run `cargo bench` from the `equix/bench` crate within Arti. This framework tests both the C and Rust implementations side-by-side.

Attacker strategies

To design a protocol and choose its parameters, we first need to understand a few high-level attacker strategies to see what we are fighting against.

Overwhelm PoW verification ("top half")

A basic attack here is the adversary spamming with bogus INTRODUCE messages so that the service does not have computing capacity to even verify the proof-of-work. This adversary tries to overwhelm the procedure in the `v1` verification algorithm section.

That's why we need the PoW algorithm to have a cheap verification time so that this attack is not possible: we explore this PoW parameter below in the section on PoW verification.

Overwhelm rendezvous capacity ("bottom half")

Given the way the introduction queue works, a very effective strategy for the attacker is to totally overwhelm the queue processing by sending more high-effort introductions than the onion service can handle at any given tick. This adversary tries to overwhelm the process of handling queued introductions.

To do so, the attacker would have to send at least 20 high-effort INTRODUCE messages every 100ms, where high-effort is a PoW which is above the estimated level of "the motivated user".

An easier attack for the adversary, is the same strategy but with INTRODUCE messages that are all above the comfortable level of "the standard user". This would block out all standard users and only allow motivated users to pass.

(This is an ECC group, so remember that scalar multiplication is the trapdoor function, and it's defined in terms of iterated point addition. See the Ed25519 paper [Reference ED25519-REFS] for a fairly clear writeup.)

Let B be the ed25519 basepoint as found in section 5 of [ED25519-B-REF]:

```
      B =
(15112221349535400772501151409588531511454012693041857206046113283949847762202,

463168356949264781694283940034751631413079938662562256157830336031652518559 60)
```

Assume B has prime order l, so lB=0. Let a master keypair be written as (a,A), where a is the private key and A is the public key (A=aB).

To derive the key for a nonce N and an optional secret s, compute the blinding factor like this:

```
          h = SHA3_256(BLIND_STRING | A | s | B | N)
          BLIND_STRING = "Derive temporary signing key" | INT_1(0)
          N = "key-blind" | INT_8(period-number) |
INT_8(period_length)
          B = "(1511[...]2202, 4631[...]5960)"

  then clamp the blinding factor 'h' according to the ed25519 spec:

          h[0] &= 248;
          h[31] &= 63;
          h[31] |= 64;

  and do the key derivation as follows:

      private key for the period:

           a' = h a mod l
           RH' = SHA-512(RH_BLIND_STRING | RH)[:32]
           RH_BLIND_STRING = "Derive temporary signing key hash input"

      public key for the period:

           A' = h A = (ha)B
```

Generating a signature of M: given a deterministic random-looking r (see EdDSA paper), take R=rB, S=r+hash(R,A',M)ah mod l. Send signature (R,S) and public key A'.

Verifying the signature: Check whether SB = R+hash(R,A',M)A'.

```
(If the signature is valid,
     SB = (r + hash(R,A',M)ah)B
        = rB + (hash(R,A',M)ah)B
        = R + hash(R,A',M)A' )

This boils down to regular Ed25519 with key pair (a', A').
```

See [KEYBLIND-REFS] for an extensive discussion on this scheme and possible alternatives. Also, see [KEYBLIND-PROOF] for a security proof of this scheme.

verification, so this ordering slightly raises the minimum effort required to perform a top-half attack.

If any of these steps fail the service MUST ignore this introduction request and abort the protocol.

In this document we call the above steps the “top half” of introduction handling. If all the steps of the “top half” have passed, then the circuit is added to the introduction queue.

Client sends its proof in an INTRO1 extension

Now that the client has an answer to the puzzle it's time to encode it into an INTRODUCE1 message. To do so the client adds an extension to the encrypted portion of the INTRODUCE1 message by using the EXTENSIONS field. The encrypted portion of the INTRODUCE1 message only gets read by the onion service and is ignored by the introduction point.

This extension includes the chosen nonce and effort in full, as well as the actual Equi-X proof. Clients provide only the first 4 bytes of the seed, enough to disambiguate between multiple recent seeds offered by the service.

This format is defined canonically as the proof-of-work extension to INTRODUCE1.

Service verifies PoW and handles the introduction

When a service receives an INTRODUCE1 with the `PROOF_OF_WORK` extension, it should check its configuration on whether proof-of-work is enabled on the service. If it's not enabled, the extension SHOULD BE ignored. If enabled, even if the suggested effort is currently zero, the service follows the procedure detailed in this section.

If the service requires the `PROOF_OF_WORK` extension but received an INTRODUCE1 message without any embedded proof-of-work, the service SHOULD consider this message as a zero-effort introduction for the purposes of the priority queue.

To verify the client's proof-of-work the service MUST do the following steps:

1. Find a valid seed `C` that starts with `POW_SEED`. Fail if no such seed exists.
2. Fail if `N = POW_NONCE` is present in the replay protection data structure.
3. Construct the *challenge string* as above by concatenating `P || ID || C || N || htonl(E)`. In this case, `E` and `N` are values provided by the client.
4. Calculate `R = ntohl(blake2b_32(P || ID || C || N || htonl(E) || S))`, as above
5. Fail if the the effort test overflows (`R * E > UINT32_MAX`).
6. Fail if Equi-X reports that the proof `S` is malformed or not applicable (`equix_verify(P || ID || C || N || htonl(E), S) != EQUIX_OK`)
7. If both the Blake2b and Equi-X tests pass, the request can be enqueued with priority `E`.

It's a minor performance optimization for services to compute the effort test before invoking `equix_verify`. Blake2b verification is cheaper than Equi-X

Appendix B: Selecting nodes [PICKNODES]

Picking introduction points Picking rendezvous points Building paths Reusing circuits

(TODO: This needs a writeup)

Appendix C: Recommendations for searching for vanity .onions [VANITY]

EDITORIAL NOTE: The author thinks that it's silly to brute-force the keyspace for a key that, when base-32 encoded, spells out the name of your website. It also feels a bit dangerous to me. If you train your users to connect to

llamanymityx4fi3l6x2gyzmtmgxjyqyorj9qsb5r543izcwymle.onion

I worry that you're making it easier for somebody to trick them into connecting to

llamanymityb4sqi0ta0tsw6uovyhwlezkcrmczeuzdvfauuemle.onion

Nevertheless, people are probably going to try to do this, so here's a decent algorithm to use.

To search for a public key with some criterion X:

Generate a random (sk,pk) pair.

While pk does not satisfy X:

```
        Add the number 8 to sk
        Add the point 8*B to pk

    Return sk, pk.
```

We add 8 and 8*B, rather than 1 and B, so that sk is always a valid Curve25519 private key, with the lowest 3 bits equal to 0.

This algorithm is safe [source: djb, personal communication] [TODO: Make sure I understood correctly!] so long as only the final (sk,pk) pair is used, and all previous values are discarded.

To parallelize this algorithm, start with an independent (sk,pk) pair generated for each independent thread, and let each search proceed independently.

See [VANITY-REFS] for a reference implementation of this vanity .onion search scheme.

3. Identity string `ID`, a 32-byte value unique to the specific onion service. This is the blinded public ID key `KP_hs_blind_id`.
4. Seed `C`, a 32-byte random value decoded from `seed-b64` above.
5. Initial nonce `N`, a 16-byte value generated using a secure random generator.

The solver itself is iterative; the following steps are repeated until they succeed:

1. Construct the *challenge string* by concatenating `P || ID || C || N || htonl(E)`.

2. Calculate a candidate proof `S` by passing this challenge to Equi-X.

   ```
   S = equix_solve(P || ID || C || N || htonl(E))
   ```

3. Calculate a 32-bit check value by interpreting a 32-bit Blake2b hash of the concatenated challenge and solution as an integer in network byte order.

   ```
   R = ntohl(blake2b_32(P || ID || C || N || htonl(E) || S))
   ```

4. Check if 32-bit multiplication of `R * E` would overflow

   If `R * E` overflows (the result would be greater than `UINT32_MAX`) the solver must retry with another nonce value. The client interprets N as a 16-byte little-endian integer, increments it by 1, and goes back to step 1.

   If there is no overflow (the result is less than or equal to `UINT32_MAX`) this is a valid solution. The client can submit final nonce `N`, effort `E`, the first 4 bytes of seed `C`, and proof `S`.

Note that the Blake2b hash includes the output length parameter in its initial state vector, so a `blake2b_32` is not equivalent to the prefix of a `blake2b_512`. We calculate the 32-bit Blake2b specifically, and interpret it in network byte order as an unsigned integer.

At the end of the above procedure, the client should have calculated a proof `S` and final nonce `N` that satisfies both the Equi-X proof conditions and the Blake2b effort test. The time taken, on average, is linearly proportional with the target effort `E` parameter.

The algorithm as described is suitable for single-threaded computation. Optionally, a client may choose multiple nonces and attempt several solutions in parallel on separate CPU cores. The specific choice of nonce is entirely up to the client, so parallelization choices like this do not impact the network protocol's interoperability at all.

Parameter descriptor

This whole protocol starts with the service encoding its parameters in a `pow-params` line within the 'encrypted' (inner) part of the v3 descriptor. The second layer plaintext format describes it canonically. The parameters offered are:

- `scheme`, always `v1` for the algorithm described here
- `seed-b64`, a periodically updated 32-byte random seed, base64 encoded
- `suggested-effort`, the latest output from the service-side effort controller
- `expiration-time`, a timestamp when we plan to replace the seed.

Seed expiration and rotation allows used nonces to expire from the anti-replay memory. At every seed rotation, a new expiration time is chosen uniformly at random from the recommended range:

- At the earliest, 105 minutes in the future
- At the latest, 2 hours in the future (15 minutes later)

The service SHOULD refresh its seed when expiration-time passes. The service SHOULD keep its previous seed in memory and accept PoWs using it to avoid race-conditions with clients that have an old seed. The service SHOULD avoid generating two consequent seeds that have a common 4 bytes prefix; see the usage of seed headings below in the introduction extension.

Client computes a solution

If a client receives a descriptor with `pow-params`, it should assume that the service is prepared to receive PoW solutions as part of the introduction protocol.

The client parses the descriptor and extracts the PoW parameters. It makes sure that the `expiration-time` has not expired. If it has, the descriptor may be out of date. Clients SHOULD fetch a fresh descriptor if the descriptor is stale and the seed is expired.

Inputs to the solver:

1. Effort `E`, the client-side effort choice made based on the server's `suggested-effort` and the client's connection attempt history. This is a 32-bit unsigned integer.
2. Constant personalization string `P`, equal to the following nul-terminated ASCII text: `"Tor hs intro v1\0"`.

Appendix E: Reserved numbers

We reserve these certificate type values for Ed25519 certificates:

```
      [08] short-term descriptor signing key, signed with blinded
           public key. (Section 2.4)
      [09] intro point authentication key, cross-certifying the
descriptor
           signing key. (Section 2.5)
      [0B] ed25519 key derived from the curve25519 intro point
encryption key,
           cross-certifying the descriptor signing key. (Section 2.5)

      Note: The value "0A" is skipped because it's reserved for the
onion key
            cross-certifying ntor identity key from proposal 228.
```

Appendix F: Hidden service directory format [HIDSERVDIR-FORMAT]

This appendix section specifies the contents of the HiddenServiceDir directory:

- "hostname" [FILE]

This file contains the onion address of the onion service.

- "private_key_ed25519" [FILE]

This file contains the private master ed25519 key of the onion service. [TODO: Offline keys]

```
- "./authorized_clients/"                  [DIRECTORY]
  "./authorized_clients/alice.auth"        [FILE]
  "./authorized_clients/bob.auth"          [FILE]
  "./authorized_clients/charlie.auth"      [FILE]
```

> Note: "restricted discovery" is called "client authorization" in the C Tor implementation

If client authorization is enabled, this directory MUST contain a ".auth" file for each authorized client. Each such file contains the public key of the respective client. The files are transmitted to the service operator by the client.

See section [RESTRICTED-DISCOVERY-MGMT] for more details and the format of the client file.

(NOTE: client authorization is implemented as of 0.3.5.1-alpha.)

Linear effort adjustment

The underlying Equi-X puzzle has an approximately fixed computational cost. Adjustable effort comes from the construction of the overlying Blake2b layer, which requires clients to test a variable number of Equi-X solutions in order to find answers which also satisfy this layer's effort constraint.

It's common for proof-of-work systems to define an exponential effort function based on a particular number of leading zero bits or equivalent. For the benefit of our effort control system, it's quite useful if we have a linear scale instead. We use the first 32 bits of a hashed version of the Equi-X solution as a uniformly distributed random value.

Conceptually we could define a function:

```
unsigned effort(uint8_t *token)
```

which takes as its argument a hashed solution, interprets it as a bitstring, and returns the quotient of dividing a bitstring of 1s by it.

So for example:

```
effort(00000001100010101101) = 11111111111111111111
                                 / 00000001100010101101
```

or the same in decimal:

```
effort(6317) = 1048575 / 6317 = 165.
```

In practice we can avoid even having to perform this division, performing just one multiply instead to see if a request's claimed effort is supported by the smallness of the resulting 32-bit hash prefix. This assumes we send the desired effort explicitly as part of each PoW solution. We do want to force clients to pick a specific effort before looking for a solution, otherwise a client could opportunistically claim a very large effort any time a lucky hash prefix comes up. Thus the effort is communicated explicitly in our protocol, and it forms part of the concatenated Equi-X challenge.

Algorithm overview

The overall scheme consists of several layers that provide different pieces of this functionality:

1. At the lowest layers, Blake2b and siphash are used as hashing and PRNG algorithms that are well suited to common 64-bit CPUs.
2. A custom hash function family, HashX, randomizes its implementation for each new seed value. These functions are tuned to utilize the pipelined integer performance on a modern 64-bit CPU. This layer provides the strongest ASIC resistance, since a hardware reimplementation would need to include a CPU-like pipelined execution unit to keep up.
3. The Equi-X layer itself builds on HashX and adds an algorithmic puzzle that's designed to be strongly asymmetric and to require RAM to solve efficiently.
4. The PoW protocol itself builds on this Equi-X function with a particular construction of the challenge input and particular constraints on the allowed Blake2b hash of the solution. This layer provides a linearly adjustable effort that we can verify.
5. At this point, all further layers are part of the common protocol. Above the level of individual PoW handshakes, the client and service form a closed-loop system that adjusts the effort of future handshakes.

Equi-X itself provides two functions that will be used in this proposal:

- `equix_solve(challenge)` which solves a puzzle instance, returning a variable number of solutions per invocation depending on the specific challenge value.
- `equix_verify(challenge, solution)` which verifies a puzzle solution quickly. Verification still depends on executing the HashX function, but far fewer times than when searching for a solution.

For the purposes of this proposal, all cryptographic algorithms are assumed to produce and consume byte strings, even if internally they operate on some other data type like 64-bit words. This is conventionally little endian order for Blake2b, which contrasts with Tor's typical use of big endian. HashX itself is configured with an 8-byte output but its input is a single 64-bit word of undefined byte order, of which only the low 16 bits are used by Equi-X in its solution output. We treat Equi-X solution arrays as byte arrays using their packed little endian 16-bit representation.

Appendix G: Managing authorized client data [RESTRICTED-DISCOVERY-MGMT]

Hidden services and clients can configure their authorized client data either using the torrc, or using the control port. This section presents a suggested scheme for configuring restricted discovery. Please see appendix [HIDSERVDIR-FORMAT] for more information about relevant hidden service files.

(NOTE: restricted discovery is implemented as of 0.3.5.1-alpha.)

G.1. Configuring restricted discovery using torrc

G.1.1. Hidden Service side configuration

> Note: "restricted discovery" is called "client authorization" in the C Tor implementation

```
     A hidden service that wants to enable client authorization, needs
to
     populate the "authorized_clients/" directory of its
HiddenServiceDir
     directory with the ".auth" files of its authorized clients.

     When Tor starts up with a configured onion service, Tor checks
its
     <HiddenServiceDir>/authorized_clients/ directory for ".auth"
files, and if
     any recognized and parseable such files are found, then client
     authorization becomes activated for that service.

  G.1.2. Service-side bookkeeping

     This section contains more details on how onion services should
be keeping
     track of their client ".auth" files.

     For the "descriptor" authentication type, the ".auth" file MUST
contain
     the x25519 public key of that client. Here is a suggested file
format:

        <auth-type>:<key-type>:<base32-encoded-public-key>

     Here is an an example:


descriptor:x25519:OM7TGIVRYMY6PFX6GAC6ATRTA5U6WW6U7A4ZNHQDI6OVL52XVV2Q

     Tor SHOULD ignore lines it does not recognize.
     Tor SHOULD ignore files that don't use the ".auth" suffix.

  G.1.3. Client side configuration

     A client who wants to register client authorization data for
onion
     services needs to add the following line to their torrc to
indicate the
     directory which hosts ".auth_private" files containing client-
side
     credentials for onion services:

         ClientOnionAuthDir <DIR>

     The <DIR> contains a file with the suffix ".auth_private" for
each onion
     service the client is authorized with. Tor should scan the
directory for
     ".auth_private" files to find which onion services require client
     authorization from this client.

     For the "descriptor" auth-type, a ".auth_private" file contains
```

Onion service proof-of-work: Scheme v1, Equi-X and Blake2b

Implementations

For our `v1` proof-of-work function we use the Equi-X asymmetric client puzzle algorithm by tevador. The concept and the C implementation were developed specifically for our use case by tevador, based on a survey of existing work and an analysis of Tor's requirements.

- Original Equi-X source repository
- Development log

Equi-X is an asymmetric PoW function based on Equihash<60,3>, using HashX as the underlying layer. It features lightning fast verification speed, and also aims to minimize the asymmetry between CPU and GPU. Furthermore, it's designed for this particular use-case and hence cryptocurrency miners are not incentivized to make optimized ASICs for it.

At this point there is no formal specification for Equi-X or the underlying HashX function. We have two actively maintained implementations of both components, which we subject to automated cross-compatibility and fuzz testing:

- A fork of tevador's implementation is maintained within the C tor repository.

  This is the `src/ext/equix` subdirectory. Currently this contains important fixes for security, portability, and testability which have not been merged upstream! This implementation is released under the LGPL license. When `tor` is built with the required `--enable-gpl` option this code will be statically linked.

- As part of Arti, a new Rust re-implementation was written based loosely on tevador's original.

  This is the `equix` crate. This implementation currently has somewhat lower verification performance than the original but otherwise offers equivalent features.

Client-imposed effort limits

There isn't a practical upper limit on effort defined by the protocol itself, but clients may choose a maximum effort limit to enforce. It may be desirable to do this in some cases to improve responsiveness, but the main reason for this limit currently is as a workaround for weak cancellation support in our implementation.

Effort values used for both initial connections and retries are currently limited to no greater than `CLIENT_MAX_POW_EFFORT` (= 10000).

TODO: This hardcoded limit should be replaced by timed limits and/or an unlimited solver with robust cancellation. This is issue 40787 in C tor.

```
the
     private x25519 key:

        <onion-address>:descriptor:x25519:<base32-encoded-privkey>

     The keypair used for client authorization is created by a third
party tool
     for which the public key needs to be transferred to the service
operator
     in a secure out-of-band way. The third party tool SHOULD add
appropriate
     headers to the private key file to ensure that users won't
accidentally
     give out their private key.

  G.2. Configuring client authorization using the control port

  G.2.1. Service side

     A hidden service also has the option to configure authorized
clients
     using the control port. The idea is that hidden service operators
can use
     controller utilities that manage their access control instead of
using
     the filesystem to register client keys.

     Specifically, we require a new control port command
ADD_ONION_CLIENT_AUTH
     which is able to register x25519/ed25519 public keys tied to a
specific
     authorized client.
      [XXX figure out control port command format]

     Hidden services who use the control port interface for client
auth need
     to perform their own key management.

  G.2.2. Client side

     There should also be a control port interface for clients to
register
     authorization data for hidden services without having to use the
     torrc. It should allow both generation of client authorization
private
     keys, and also to import client authorization data provided by a
hidden
     service

     This way, Tor Browser can present "Generate client auth keys" and
"Import
     client auth keys" dialogs to users when they try to visit a
hidden service
     that is protected by client authorization.
```

```
     Specifically, we require two new control port commands:
                  IMPORT_ONION_CLIENT_AUTH_DATA
                  GENERATE_ONION_CLIENT_AUTH_DATA
     which import and generate client authorization data respectively.

     [XXX how does key management work here?]
     [XXX what happens when people use both the control port interface
and the
          filesystem interface?]
```

INTRODUCE_ACK message is not end-to-end (it's from the introduction point to the client) and hence it does not allow the service to inform the client that the rendezvous is never gonna occur.

From the client's perspective there's no way to attribute this failure to the service itself rather than the introduction point, so error accounting is performed separately for each introduction-point. Prior mechanisms will discard an introduction point that's required too many retries.

Clients handling timeouts

Alice can fail to reach the onion service if her introduction request gets trimmed off the priority queue when enqueueing new requests, or if the service does not get through its priority queue in time and the connection times out.

This section presents a heuristic method for the client getting service even in such scenarios.

If the rendezvous request times out, the client SHOULD fetch a new descriptor for the service to make sure that it's using the right suggested-effort for the PoW and the right PoW seed. If the fetched descriptor includes a new suggested effort or seed, it should first retry the request with these parameters.

TODO: This is not actually implemented yet, but we should do it. How often should clients at most try to fetch new descriptors? Determined by a consensus parameter? This change will also allow clients to retry effectively in cases where the service has just been reconfigured to enable PoW defenses.

Every time the client retries the connection, it will count these failures per-introduction-point. These counts of previous retries are combined with the service's `suggested_effort` when calculating the actual effort to spend on any individual request to a service that advertises PoW support, even when the currently advertised `suggested_effort` is zero.

On each retry, the client modifies its solver effort:

1. If the effort is below `CLIENT_POW_EFFORT_DOUBLE_UNTIL` (= 1000) it will be doubled.
2. Otherwise, multiply the effort by `CLIENT_POW_RETRY_MULTIPLIER` (= 1.5).
3. Constrain the effort to no less than `CLIENT_MIN_RETRY_POW_EFFORT` (= 8). Note that this limit is specific to retries only. Clients may use a lower effort for their first connection attempt.
4. Apply the maximum effort limit described below.

Over time, this will continue to decrease our effort suggestion any time the service is fully processing its request queue. If the queue stays empty, the effort suggestion decreases to zero and clients should no longer submit a proof-of-work solution with their first connection attempt.

It's worth noting that the `suggested_effort` is not a hard limit to the efforts that are accepted by the service, and it's only meant to serve as a guideline for clients to reduce the number of unsuccessful requests that get to the service. When adding requests to the queue, services do accept valid solutions with efforts higher or lower than the published values from `pow-params`.

Updating descriptor with new suggested effort

The service descriptors may be updated for multiple reasons including introduction point rotation common to all v3 onion services, scheduled seed rotations like the one described for `v1` parameters, and updates to the effort suggestion. Even though the internal effort value updates on a regular timer, we avoid propagating those changes into the descriptor and the HSDir hosts unless there is a significant change.

If the PoW params otherwise match but the seed has changed by less than 15 percent, services SHOULD NOT upload a new descriptor.

Client-side effort control

Clients are responsible for making their own effort adjustments in response to connection trouble, to allow the system a chance to react before the service has published new effort values. This is an important tool to uphold UX expectations without relying on excessively frequent updates through the HSDir.

TODO: This is the weak link in user experience for our current implementation. The C tor implementation does not detect and retry onion service connections as reliably as we would like. Currently our best strategy to improve retry behavior is the Arti rewrite.

Failure ambiguity

The first challenge in reacting to failure, in our case, is to even accurately and quickly understand when a failure has occurred.

This proposal introduces a bunch of new ways where a legitimate client can fail to reach the onion service. Furthermore, there is currently no end-to-end way for the onion service to inform the client that the introduction failed. The

Appendix F: Two methods for managing revision counters

Implementations MAY generate revision counters in any way they please, so long as they are monotonically increasing over the lifetime of each blinded public key. But to avoid fingerprinting, implementors SHOULD choose a strategy also used by other Tor implementations. Here we describe two, and additionally list some strategies that implementors should NOT use.

F.1. Increment-on-generation

This is the simplest strategy, and the one used by Tor through at least version 0.3.4.0-alpha.

Whenever using a new blinded key, the service records the highest revision counter it has used with that key. When generating a descriptor, the service uses the smallest non-negative number higher than any number it has already used.

In other words, the revision counters under this system start fresh with each blinded key as 0, 1, 2, 3, and so on.

F.2. Encrypted time in period

This scheme is what we recommend for situations when multiple service instances need to coordinate their revision counters, without an actual coordination mechanism.

Let T be the number of seconds that have elapsed since the start time of the SRV protocol run that corresponds to this descriptor, plus 1. (T must be at least 1.)

Let S be a per-time-period secret that all the service providers share. (C tor and arti use `S = KS_hs_blind_id`; when `KS_hs_blind_id` is not available, implementations may use `S = KS_hs_desc_sign`.)

Let K be an AES-256 key, generated as

```
K = SHA3_256("rev-counter-generation" | S)
```

Use `K`, and AES in counter mode with IV=0, to generate a stream of `T * 2` bytes. Consider these bytes as a sequence of T 16-bit little-endian words. Add these words.

Let the sum of these words, plus T, be the revision counter.

> (We include T in the sum so that every increment in T adds at least one to the output.)

Cryptowiki attributes roughly this scheme to G. Bebek in:

> G. Bebek. Anti-tamper database research: Inference control techniques. Technical Report EECS 433 Final Report, Case Western Reserve University, November 2002.

Although we believe it is suitable for use in this application, it is not a perfect order-preserving encryption algorithm (and all order-preserving encryption has weaknesses). Please think twice before using it for anything else.

(This scheme can be optimized pretty easily by caching the encryption of `X*1`, `X*2`, `X*3`, etc for some well chosen `X`.)

For a slow reference implementation that can generate test vectors, see `src/test/ope_ref.py` in the Tor source repository.

> **Note:**
>
> Some onion service operators have historically relied upon the behavior of this OPE scheme to provide a kind of ersatz load-balancing: they run multiple onion services, each with the the same `KP_hs_id`. The onion services choose different introduction points, and race each other to publish their HSDescs.
>
> It's probably better to use Onionbalance or a similar tool for this purpose. Nonetheless, onion services implemenentations MAY choose to implement this particular OPE scheme exactly in order to provide interoperability with C tor in this use case.

F.X. Some revision-counter strategies to avoid

Though it might be tempting, implementations SHOULD NOT use the current time or the current time within the period directly as their revision counter – doing so leaks their view of the current time, which can be used to link the onion service to other services run on the same host.

TODO: `HS_UPDATE_PERIOD` is hardcoded to 300 (5 minutes) currently, but it should be configurable in some way. Is it more appropriate to use the service's torrc here or a consensus parameter?

Per-update-period service state

During each update period, the service maintains some state:

1. `TOTAL_EFFORT`, a sum of all effort values for rendezvous requests that were successfully validated and enqueued.
2. `REND_HANDLED`, a count of rendezvous requests that were actually launched. Requests that made it to dequeueing but were too old to launch by then are not included.
3. `HAD_QUEUE`, a flag which is set if at any time in the update period we saw the priority queue filled with more than a minimum amount of work, greater than we would expect to process in approximately 1/4 second using the configured dequeue rate.
4. `MAX_TRIMMED_EFFORT`, the largest observed single request effort that we discarded during the update period. Requests are discarded either due to age (timeout) or during culling events that discard the bottom half of the entire queue when it's too full.

Service AIMD conditions

At the end of each update period, the service may decide to increase effort, decrease effort, or make no changes, based on these accumulated state values:

1. If `MAX_TRIMMED_EFFORT` > our previous internal `suggested_effort`, always INCREASE. Requests that follow our latest advice are being dropped.
2. If the `HAD_QUEUE` flag was set and the queue still contains at least one item with effort >= our previous internal `suggested_effort`, INCREASE. Even if we haven't yet reached the point of dropping requests, this signal indicates that our latest suggestion isn't high enough and requests will build up in the queue.
3. If neither condition 1 or 2 are taking place and the queue is below a level we would expect to process in approximately 1/4 second, choose to DECREASE.
4. If none of these conditions match, the `suggested_effort` is unchanged.

When we INCREASE, the internal `suggested_effort` is increased to either its previous value + 1, or (`TOTAL_EFFORT` / `REND_HANDLED`), whichever is larger.

When we DECREASE, the internal `suggested_effort` is scaled by 2/3rds.

Motivated users can spend a high amount of effort in their PoW computation, which should guarantee access to the service given reasonable adversary models.

An effective effort control algorithm will improve reachability and UX by suggesting values that reduce overall service load to tolerable values while also leaving users with a tolerable overall delay.

The service starts with a default suggested-effort value of 0, which keeps the PoW defenses dormant until we notice signs of queue overload.

The entire process of determining effort can be thought of as a set of multiple coupled feedback loops. Clients perform their own effort adjustments via timeout retry atop a base effort suggested by the service. That suggestion incorporates the service's control adjustments atop a base effort calculated using a sum of currently-queued client effort.

Each feedback loop has an opportunity to cover different time scales. Clients can make adjustments at every single circuit creation request, whereas services are limited by the extra load that frequent updates would place on HSDir nodes.

In the combined client/service system these client-side increases are expected to provide the most effective quick response to an emerging DoS attack. After early clients increase the effort using timeouts, later clients benefit from the service detecting this increased queued effort and publishing a larger suggested effort.

Effort increases and decreases both have a cost. Increasing effort will make the service more expensive to contact, and decreasing effort makes new requests likely to become backlogged behind older requests. The steady state condition is preferable to either of these side-effects, but ultimately it's expected that the control loop always oscillates to some degree.

Service-side effort control

Services keep an internal suggested effort target which updates on a regular periodic timer in response to measurements made on queue behavior in the previous update period. These internal effort changes can optionally trigger client-visible descriptor changes when the difference is great enough to warrant republication to the HSDir.

This evaluation and update period is referred to as `HS_UPDATE_PERIOD`. The service-side effort control loop takes inspiration from TCP congestion control's additive increase / multiplicative decrease approach, but unlike a typical AIMD this algorithm is fixed-rate and doesn't update immediately in response to events.

Similarly, implementations SHOULD NOT let the revision counter increase forever without resetting it – doing so links the service across changes in the blinded public key.

Appendix G: Test vectors

G.1. Test vectors for hs-ntor / NTOR-WITH-EXTRA-DATA

Adding introductions to the introduction queue

When PoW is enabled and an introduction request includes a verified proof, the service queues each request in a data structure sorted by effort. Requests including no proof at all MUST be assigned an effort of zero. Requests with a proof that fails to verify MUST be rejected and not enqueued.

Services MUST check whether the queue is overfull when adding to it, not just when processing requests. Floods of low-effort and zero-effort introductions need to be efficiently discarded when the queue is growing faster than it's draining.

The C implementation chooses a maximum number of queued items based on its configured dequeue rate limit multiplied by the circuit timeout. In effect, items past this threshold are expected not to be reachable by the time they will timeout. When this limit is exceeded, the queue experiences a mass trim event where the lowest effort half of all items are discarded.

Handling queued introductions

When deciding which introduction request to consider next, the service chooses the highest available effort. When efforts are equivalent, the oldest queued request is chosen.

The service should handle introductions only by pulling from the introduction queue. We call this part of introduction handling the "bottom half" because most of the computation happens in this stage.

For more on how we expect such a system to work in Tor, see the scheduler analysis and discussion section.

Effort control

Overall strategy for effort determination

Denial-of-service is a dynamic problem where the attacker's capabilities constantly change, and hence we want our proof-of-work system to be dynamic and not stuck with a static difficulty setting. Instead of forcing clients to go below a static target configured by the service operator, we ask clients to "bid" using their PoW effort. Effectively, a client gets higher priority the higher effort they put into their proof-of-work. Clients automatically increase their bid when retrying, and services regularly offer a suggested starting point based on the recent queue status.

these cases, the proof-of-work subsystem can be dormant but still provide the necessary parameters for clients to voluntarily provide effort in order to get better placement in the priority queue.

The protocol involves the following major steps:

1. Service encodes PoW parameters in descriptor: `pow-params` in the second layer plaintext format.
2. Client fetches descriptor and begins solving. Currently this must use the `v1` solver algorithm.
3. Client finishes solving and sends results using the proof-of-work extension to INTRODUCE1.
4. Service verifies the proof and queues an introduction based on proven effort. This currently uses the `v1` verify algorithm only.
5. Requests are continuously drained from the queue, highest effort first, subject to multiple constraints on speed. See below for more on handling queued requests.

Replay protection

The service MUST NOT accept introduction requests with the same (seed, nonce) tuple. For this reason a replay protection mechanism must be employed.

The simplest way is to use a hash table to check whether a (seed, nonce) tuple has been used before for the active duration of a seed. Depending on how long a seed stays active this might be a viable solution with reasonable memory/time overhead.

If there is a worry that we might get too many introductions during the lifetime of a seed, we can use a Bloom filter or similar as our replay cache mechanism. A probabilistic filter means that we will potentially flag some connections as replays even if they are not, with this false positive probability increasing as the number of entries increase. With the right parameter tuning this probability should be negligible, and dropped requests will be retried by the client.

The introduction queue

When proof-of-work is enabled for a service, that service diverts all incoming introduction requests to a priority queue system rather than handling them immediately.

```
    Here is a set of test values for the hs-ntor handshake, called
    [NTOR-WITH-EXTRA-DATA] in this document.  They were generated by
    instrumenting Tor's code to dump the values for an INTRODUCE/
RENDEZVOUS
    handshake, and then by running that code on a Chutney network.

    We assume an onion service with:

      KP_hs_ipd_sid = 34E171E4358E501BFF21ED907E96AC6B
                      FEF697C779D040BBAF49ACC30FC5D21F
      KP_hss_ntor   = 8E5127A40E83AABF6493E41F142B6EE3
                      604B85A3961CD7E38D247239AFF71979
      KS_hss_ntor   = A0ED5DBF94EEB2EDB3B514E4CF6ABFF6
                      022051CC5F103391F1970A3FCD15296A
      N_hs_subcred  = 0085D26A9DEBA252263BF0231AEAC59B
                      17CA11BAD8A218238AD6487CBAD68B57

    The client wants to make in INTRODUCE request.  It generates
    the following header (everything before the ENCRYPTED portion)
    of its INTRODUCE1 message:

      H =
0000000000000000000000000000000000000002002034E171E4358E501BFF
          21ED907E96AC6BFEF697C779D040BBAF49ACC30FC5D21F00

    It generates the following plaintext body to encrypt.  (This
    is the "decrypted plaintext body" from [PROCESS_INTRO2].

      P =
6BD364C12638DD5C3BE23D76ACA05B04E6CE932C0101000100200DE6130E4FCA

C4EDDA24E21220CC3EADAE403EF6B7D11C8273AC71908DE565450300067F0000

0113890214F823C4F8CC085C792E0AEE0283FE00AD7520B37D0320728D5DF39B

7B7077A0118A900FF4456C382F0041300ACF9C58E51C392795EF870000000000

0000000000000000000000000000000000000000000000000000000000000000
          0000000000000000000000000000000000000000000000000000000000000000

      (Note! This should in fact be padded to be longer; when these
      test vectors were generated, the target INTRODUCE1 length in C
      Tor was needlessly short.)

    The client now begins the hs-ntor handshake.  It generates
    a curve25519 keypair:

      x = 60B4D6BF5234DCF87A4E9D7487BDF3F4
          A69B6729835E825CA29089CFDDA1E341
      X = BF04348B46D09AED726F1D66C618FDEA
          1DE58E8CB8B89738D7356A0C59111D5D

    Then it calculates:
```

```
  ENC_KEY = 9B8917BA3D05F3130DACCE5300C3DC27
            F6D012912F1C733036F822D0ED238706
  MAC_KEY = FC4058DA59D4DF61E7B40985D122F502
            FD59336BC21C30CAF5E7F0D4A2C38FD5

With these, it encrypts the plaintext body P with ENC_KEY, getting
an encrypted value C.  It computes MAC(MAC_KEY, H | X | C),
getting a MAC value M.  It then assembles the final INTRODUCE1
body as H | X | C | M:

  0000000000000000000000000000000000000000002002034E171E4358E501BFF
  21ED907E96AC6BFEF697C779D040BBAF49ACC30FC5D21F00BF04348B46D09AED
  726F1D66C618FDEA1DE58E8CB8B89738D7356A0C59111D5DADBECCCB38E37830
  4DCC179D3D9E437B452AF5702CED2CCFEC085BC02C4C175FA446525C1B9D5530
  563C362FDFFB802DAB8CD9EBC7A5EE17DA62E37DEEB0EB187FBB48C63298B0E8
  3F391B7566F42ADC97C46BA7588278273A44CE96BC68FFDAE31EF5F0913B9A9C
  7E0F173DBC0BDDCD4ACB4C4600980A7DDD9EAEC6E7F3FA3FC37CD95E5B8BFB3E
  35717012B78B4930569F895CB349A07538E42309C993223AEA77EF8AEA64F25D
  DEE97DA623F1AEC0A47F150002150455845C385E5606E41A9A199E7111D54EF2
  D1A51B7554D8B3692D85AC587FB9E69DF990EFB776D8
```

Later the service receives that body in an INTRODUCE2 message. It processes it according to the hs-ntor handshake, and recovers the client's plaintext P. To continue the hs-ntor handshake, the service chooses a curve25519 keypair:

```
  y = 68CB5188CA0CD7924250404FAB54EE13
      92D3D2B9C049A2E446513875952F8F55
  Y = 8FBE0DB4D4A9C7FF46701E3E0EE7FD05
      CD28BE4F302460ADDEEC9E93354EE700

From this and the client's input, it computes:

  AUTH_INPUT_MAC = 4A92E8437B8424D5E5EC279245D5C72B
                   25A0327ACF6DAF902079FCB643D8B208
  NTOR_KEY_SEED  = 4D0C72FE8AFF35559D95ECC18EB5A368
                   83402B28CDFD48C8A530A5A3D7D578DB
```

The service sends back Y | AUTH_INPUT_MAC in its RENDEZVOUS1 message body. From these, the client finishes the handshake, validates AUTH_INPUT_MAC, and computes the same NTOR_KEY_SEED.

Now that both parties have the same NTOR_KEY_SEED, they can derive the shared key material they will use for their circuit.

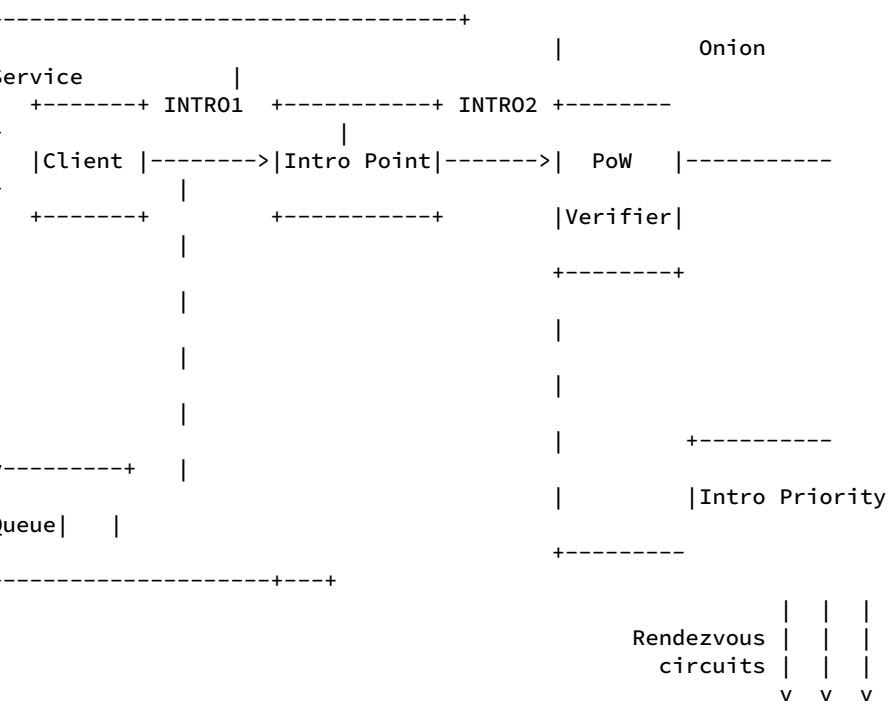
Common protocol

We have made an effort to split the design of the proof-of-work subsystem into an algorithm-specific piece that can be upgraded, and a core protocol that provides queueing and effort adjustment.

Currently there is only one versioned subprotocol defined:

- Version 1, Equi-X and Blake2b

Overview



The proof-of-work scheme specified in this document takes place during the introduction phase of the onion service protocol.

The system described in this proposal is not meant to be on all the time, and it can be entirely disabled for services that do not experience DoS attacks.

When the subsystem is enabled, suggested effort is continuously adjusted and the computational puzzle can be bypassed entirely when the effort reaches zero. In

- "The motivated user"

  This is a user that really wants to reach their destination. They don't care about the journey; they just want to get there. They know what's going on; they are willing to make their computer do expensive multi-minute PoW computations to get where they want to be.

- "The mobile user"

  This is a motivated user on a mobile phone. Even tho they want to read the news article, they don't have much leeway on stressing their machine to do more computation.

We hope that this proposal will allow the motivated user to always connect where they want to connect to, and also give more chances to the other user groups to reach the destination.

The DoS Catch-22

This proposal is not perfect and it does not cover all the use cases. Still, we think that by covering some use cases and giving reachability to the people who really need it, we will severely demotivate the attackers from continuing the DoS attacks and hence stop the DoS threat all together. Furthermore, by increasing the cost to launch a DoS attack, a big class of DoS attackers will disappear from the map, since the expected ROI will decrease.

Proof of Work for onion service introduction

The overall denial-of-service prevention strategies in Tor are described in the Denial-of-service prevention mechanisms in Tor document. This document describes one specific mitigation, the proof-of-work client puzzle for onion service introduction.

This was originally proposal 327, A First Take at PoW Over Introduction Circuits authored by George Kadianakis, Mike Perry, David Goulet, and tevador.

Motivation

See the denial-of-service overview for the big-picture view. Here we are focusing on a mitigation for attacks on one specific resource: onion service introductions.

Attackers can generate low-effort floods of introductions which cause the onion service and all involved relays to perform a disproportionate amount of work, leading to a denial-of-service opportunity. This proof-of-work scheme intends to make introduction floods unattractive to attackers, reducing the network-wide impact of this activity.

Previous to this work, our attempts at limiting the impact of introduction flooding DoS attacks on onion services has been focused on horizontal scaling with Onionbalance, optimizing the CPU usage of Tor and applying rate limiting. While these measures move the goalpost forward, a core problem with onion service DoS is that building rendezvous circuits is a costly procedure both for the service and for the network.

For more information on the limitations of rate-limiting when defending against DDoS, see `draft-nygren-tls-client-puzzles-02`.

If we ever hope to have truly reachable global onion services, we need to make it harder for attackers to overload the service with introduction requests. This proposal achieves this by allowing onion services to specify an optional dynamic proof-of-work scheme that its clients need to participate in if they want to get served.

With the right parameters, this proof-of-work scheme acts as a gatekeeper to block amplification attacks by attackers while letting legitimate clients through.

Related work

For a similar concept, see the three internet drafts that have been proposed for defending against TLS-based DDoS attacks using client puzzles:

- `draft-nygren-tls-client-puzzles-02`
- `draft-nir-tls-puzzles-00`
- `draft-ietf-ipsecme-ddos-protection-10`

Threat model

Attacker profiles

This mitigation is written to thwart specific attackers. The current protocol is not intended to defend against all and every DoS attack on the Internet, but there are adversary models we can defend against.

Let's start with some adversary profiles:

- "The script-kiddie"

  The script-kiddie has a single computer and pushes it to its limits. Perhaps it also has a VPS and a pwned server. We are talking about an attacker with total access to 10 GHz of CPU and 10 GB of RAM. We consider the total cost for this attacker to be zero $.

- "The small botnet"

  The small botnet is a bunch of computers lined up to do an introduction flooding attack. Assuming 500 medium-range computers, we are talking about an attacker with total access to 10 THz of CPU and 10 TB of RAM. We consider the upfront cost for this attacker to be about $400.

- "The large botnet"

  The large botnet is a serious operation with many thousands of computers organized to do this attack. Assuming 100k medium-range computers, we are talking about an attacker with total access to 200 THz of CPU and 200 TB of RAM. The upfront cost for this attacker is about $36k.

We hope that this proposal can help us defend against the script-kiddie attacker and small botnets. To defend against a large botnet we would need more tools at our disposal (see the discussion on future designs).

User profiles

We have attackers and we have users. Here are a few user profiles:

- "The standard web user"

  This is a standard laptop/desktop user who is trying to browse the web. They don't know how these defences work and they don't care to configure or tweak them. If the site doesn't load, they are gonna close their browser and be sad at Tor. They run a 2GHz computer with 4GB of RAM.