The HANDSHAKE_AUTH field contains the MAC of all earlier fields in the message using as its key the shared per-circuit material ("KH") generated during the circuit extension protocol. It prevents replays of ESTABLISH_INTRO messages.

> Note that this usage of "KH" is ad-hoc, and should not be emulated elsewhere: it lacks any protocol-specific personalization strings. For a better design, see proposal 357
>
> Note that in our obsolete TAP circuit-extension handshake, "KH" was sent in the clear. TAP is obsolete, and unsupported on the network, and so it cannot be used with v3 onion services.

SIG_LEN is the length of the signature.

SIG is a signature, using AUTH_KEY, of all contents of the message, up to but not including SIG_LEN and SIG. These contents are prefixed with the string "Tor establish-intro cell v1".

> (Note that this string is *sic*; it predates our efforts to distinguish cells from relay messages.)

Upon receiving an ESTABLISH_INTRO message, a Tor node first decodes the key and the signature, and checks the signature. The node must reject the ESTABLISH_INTRO message and destroy the circuit in these cases:

```
* If the key type is unrecognized
* If the key is ill-formatted
* If the signature is incorrect
* If the HANDSHAKE_AUTH value is incorrect

* If the circuit is already a rendezvous circuit.
* If the circuit is already an introduction circuit.
  [TODO: some scalability designs fail there.]
* If the key is already in use by another circuit.
```

Otherwise, the node must associate the key with the circuit, for use later in INTRODUCE1 messages.

Denial-of-Service defense extension (DOS_PARAMS)

The `DOS_PARAMS` extension in ESTABLISH_INTRO is used to send Denial-of-Service (DoS) parameters to the introduction point in order for it to apply them for the introduction circuit.

```
const CIRCPAD_COMMAND_STOP = 1;
const CIRCPAD_COMMAND_START = 2;

const CIRCPAD_RESPONSE_OK = 1;
const CIRCPAD_RESPONSE_ERR = 2;

const CIRCPAD_MACHINE_CIRC_SETUP = 1;

struct circpad_negotiate {
  u8 version IN [0];
  u8 command IN [CIRCPAD_COMMAND_START, CIRCPAD_COMMAND_STOP];

  u8 machine_type IN [CIRCPAD_MACHINE_CIRC_SETUP];

  u8 unused; // Formerly echo_request

  u32 machine_ctr;
};
```

When a client wants to start a circuit padding machine, it first checks that the desired destination hop advertises the appropriate subprotocol for that machine. It then sends a PADDING_NEGOTIATE message to that hop with command=CIRCPAD_COMMAND_START, and machine_type=CIRCPAD_MACHINE_CIRC_SETUP (for the circ setup machine, the destination hop is the second hop in the circuit). The machine_ctr is the count of which machine instance this is on the circuit. It is used to disambiguate shutdown requests.

When a relay receives a PADDING_NEGOTIATE message, it checks that it supports the requested machine, and sends a PADDING_NEGOTIATED message, which is formatted in the body of a relay message with command number 42 (see tor-spec.txt section 6.1), as follows:

```
struct circpad_negotiated {
  u8 version IN [0];
  u8 command IN [CIRCPAD_COMMAND_START, CIRCPAD_COMMAND_STOP];
  u8 response IN [CIRCPAD_RESPONSE_OK, CIRCPAD_RESPONSE_ERR];

  u8 machine_type IN [CIRCPAD_MACHINE_CIRC_SETUP];

  u32 machine_ctr;
};
```

If the machine is supported, the response field will contain CIRCPAD_RESPONSE_OK. If it is not, it will contain CIRCPAD_RESPONSE_ERR.

Either side may send a CIRCPAD_COMMAND_STOP to shut down the padding machines (clients MUST only send circpad_negotiate, and relays MUST only send circpad_negotiated for this purpose).

If the machine_ctr does not match the current machine instance count on the circuit, the command is ignored.

Circuit Padding Machine Message Management

Clients MAY send padding cells towards the relay before receiving the circpad_negotiated response, to allow for outbound cover traffic before negotiation completes.

Clients MAY send another PADDING_NEGOTIATE message before receiving the circpad_negotiated response, to allow for rapid machine changes.

Relays MUST NOT send padding cells or PADDING_NEGOTIATE messages unless a padding machine is active. Any padding cells or padding-related messages that arrive at the client from unexpected relay sources are protocol violations, and clients MAY immediately tear down such circuits to avoid side channel risk.

Obfuscating client-side onion service circuit setup

The circuit padding currently deployed in Tor attempts to hide client-side onion service circuit setup. Service-side setup is not covered, because doing so would involve significantly more overhead, and/or require interaction with the application layer.

The approach taken aims to make client-side introduction and rendezvous circuits match the cell direction sequence and cell count of 3 hop general circuits used for normal web traffic, for the first 10 cells only. The lifespan of introduction circuits is also made to match the lifespan of general circuits.

Note that inter-arrival timing is not obfuscated by this defense.

Common general circuit construction sequences

Most general Tor circuits used to surf the web or download directory information start with the following 6-cell relay cell sequence (cells surrounded in [brackets] are outgoing, the others are incoming):

```
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]
HANDSHAKE_AUTH   [MAC_LEN bytes]
SIG_LEN          [2 bytes]
SIG              [SIG_LEN bytes]
```

The AUTH_KEY_TYPE field indicates the type of the introduction point authentication key and the type of the MAC to use in HANDSHAKE_AUTH. Recognized types are:

```
[00, 01] -- Reserved for legacy introduction messages; see
            [LEGACY_EST_INTRO below]
[02] -- Ed25519; SHA3-256.
```

The AUTH_KEY_LEN field determines the length of the AUTH_KEY field. The AUTH_KEY field contains the public introduction point authentication key, KP_hs_ipt_sid.

The EXT_FIELD_TYPE, EXT_FIELD_LEN, EXT_FIELD entries are reserved for extensions to the introduction protocol. Extensions with unrecognized EXT_FIELD_TYPE values must be ignored. (`EXT_FIELD_LEN` may be zero, in which case EXT_FIELD is absent.)

```
   Unless otherwise specified in the documentation for an extension
type:
      * Each extension type SHOULD be sent only once in a message.
      * Parties MUST ignore any occurrences all occurrences of an
extension
        with a given type after the first such occurrence.
      * Extensions SHOULD be sent in numerically ascending order by
type.
   (The above extension sorting and multiplicity rules are only
defaults;
   they may be overridden in the descriptions of individual
extensions.)
```

The following extensions are currently defined:

EXT_FIELD_TYPE	Name
[01]	DOS_PARAMS

The introduction protocol

The introduction protocol proceeds in three steps.

First, a hidden service host builds an anonymous circuit to a Tor node and registers that circuit as an introduction point.

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

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

        [After 'First' and before 'Second', the hidden service
publishes its
        introduction points and associated keys, and the client
fetches
        them as described in section [HSDIR] above.]
```

Second, a client builds an anonymous circuit to the introduction point, and sends an introduction request.

Third, the introduction point relays the introduction request along the introduction circuit to the hidden service host, and acknowledges the introduction request to the client.

Registering an introduction point

Extensible ESTABLISH_INTRO protocol

When a hidden service is establishing a new introduction point, it sends an ESTABLISH_INTRO message with the following contents:

[EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [BEGIN] -> CONNECTED

When this is done, the client has established a 3-hop circuit and also opened a stream to the other end. Usually after this comes a series of DATA message that either fetches pages, establishes an SSL connection or fetches directory information:

[DATA] -> [DATA] -> DATA -> DATA...(inbound cells continue)

The above stream of 10 relay cells defines the grand majority of general circuits that come out of Tor browser during our testing, and it's what we use to make introduction and rendezvous circuits blend in.

Please note that in this section we only investigate relay cells and not connection-level cells like CREATE/CREATED or AUTHENTICATE/etc. that are used during the link-layer handshake. The rationale is that connection-level cells depend on the type of guard used and are not an effective fingerprint for a network/guard-level adversary.

Client-side onion service introduction circuit obfuscation

Two circuit padding machines work to hide client-side introduction circuits: one machine at the origin, and one machine at the second hop of the circuit. Each machine sends padding towards the other. The padding from the origin-side machine terminates at the second hop and does not get forwarded to the actual introduction point.

From Section 3.3.1 above, most general circuits have the following initial relay cell sequence (outgoing cells marked in [brackets]):

```
  [EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [BEGIN] ->
CONNECTED
    -> [DATA] -> [DATA] -> DATA -> DATA...(inbound data cells
continue)

  Whereas normal introduction circuits usually look like:

  [EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [EXTEND2] ->
EXTENDED2
    -> [INTRO1] -> INTRODUCE_ACK
```

This means that up to the sixth cell (first line of each sequence above), both general and intro circuits have identical cell sequences. After that we want to mimic the second line sequence of

-> [DATA] -> [DATA] -> DATA -> DATA...(inbound data cells continue)

We achieve this by starting padding INTRODUCE1 has been sent. With padding negotiation cells, in the common case of the second line looks like:

-> [INTRO1] -> [PADDING_NEGOTIATE] -> PADDING_NEGOTIATED -> INTRO_ACK

Then, the middle node will send between INTRO_MACHINE_MINIMUM_PADDING (7) and INTRO_MACHINE_MAXIMUM_PADDING (10) cells, to match the "...(inbound data cells continue)" portion of the trace (aka the rest of an HTTPS response body).

We also set a special flag which keeps the circuit open even after the introduction is performed. With this feature the circuit will stay alive for the same duration as normal web circuits before they expire (usually 10 minutes).

Client-side rendezvous circuit hiding

Following a similar argument as for intro circuits, we are aiming for padded rendezvous circuits to blend in with the initial cell sequence of general circuits which usually look like this:

```
   [EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [BEGIN] ->
 CONNECTED
      -> [DATA] -> [DATA] -> DATA -> DATA...(incoming cells continue)

   Whereas normal rendezvous circuits usually look like:

   [EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [EST_REND] ->
 REND_EST
      -> REND2 -> [BEGIN]
```

This means that up to the sixth cell (the first line), both general and rend circuits have identical cell sequences.

After that we want to mimic a [DATA] -> [DATA] -> DATA -> DATA sequence.

With padding negotiation right after the REND_ESTABLISHED, the sequence becomes:

```
   [EXTEND2] -> EXTENDED2 -> [EXTEND2] -> EXTENDED2 -> [EST_REND] ->
 REND_EST
      -> [PADDING_NEGOTIATE] -> [DROP] -> PADDING_NEGOTIATED -> DROP...

   After which normal application DATA-bearing cells continue on the
 circuit.
```

```
       SALT = 16 bytes from SHA3_256(random), changes each time we
 rebuild the
              descriptor even if the content of the descriptor hasn't
 changed.
              (So that we don't leak whether the intro point list etc.
 changed)

       secret_input = SECRET_DATA | N_hs_subcred |
 INT_8(revision_counter)

       keys = SHAKE256_KDF(secret_input | salt | STRING_CONSTANT,
 S_KEY_LEN + S_IV_LEN + MAC_KEY_LEN)

       SECRET_KEY = first S_KEY_LEN bytes of keys
       SECRET_IV  = next S_IV_LEN bytes of keys
       MAC_KEY    = last MAC_KEY_LEN bytes of keys

   The encrypted data has the format:

       SALT       hashed random bytes from above  [16 bytes]
       ENCRYPTED  The ciphertext                  [variable]
       MAC        D_MAC of both above fields      [32 bytes]

   The final encryption format is ENCRYPTED =
 STREAM(SECRET_IV,SECRET_KEY) XOR Plaintext .

   Where D_MAC = SHA3_256(mac_key_len | MAC_KEY | salt_len | SALT |
 ENCRYPTED)
   and
    mac_key_len = htonll(len(MAC_KEY))
   and
    salt_len = htonll(len(SALT)).
```

Number of introduction points

This section defines how many introduction points an hidden service descriptor can have at minimum, by default and the maximum:

Minimum: 0 - Default: 3 - Maximum: 20

A value of 0 would means that the service is still alive but doesn't want to be reached by any client at the moment. Note that the descriptor size increases considerably as more introduction points are added.

The reason for a maximum value of 20 is to give enough scalability to tools like OnionBalance to be able to load balance up to 120 servers (20 x 6 HSDirs) but also in order for the descriptor size to not overwhelmed hidden service directories with user defined values that could be gigantic.

```
        "legacy-key-cert" NL certificate NL

          [None or at most once per introduction point]
          [This field is obsolete and should never be generated; it
           is included for historical reasons only.]

          MUST be present if "legacy-key" is present.

          The certificate is a proposal 220 RSA->Ed cross-certificate
wrapped
          in "-----BEGIN CROSSCERT-----" armor, cross-certifying the
RSA
          public key found in "legacy-key" using the descriptor
signing key.
```

To remain compatible with future revisions to the descriptor format, clients should ignore unrecognized lines in the descriptor. Other encryption and authentication key formats are allowed; clients should ignore ones they do not recognize.

Clients who manage to extract the introduction points of the hidden service can proceed with the introduction protocol as specified in [INTRO-PROTOCOL].

Compatibility note: At least some versions of OnionBalance do not include a final newline when generating this inner plaintext section; other implementations MUST accept this section even if it is missing its final newline.

Deriving hidden service descriptor encryption keys

In this section we present the generic encryption format for hidden service descriptors. We use the same encryption format in both encryption layers, hence we introduce two customization parameters SECRET_DATA and STRING_CONSTANT which vary between the layers.

The SECRET_DATA parameter specifies the secret data that are used during encryption key generation, while STRING_CONSTANT is merely a string constant that is used as part of the KDF.

Here is the key generation logic:

Hence this way we make rendezvous circuits look like general circuits up till the end of the circuit setup.

After that our machine gets deactivated, and we let the actual rendezvous circuit shape the traffic flow. Since rendezvous circuits usually imitate general circuits (their purpose is to surf the web), we can expect that they will look alike.

Circuit setup machine overhead

For the intro circuit case, we see that the origin-side machine just sends a single PADDING_NEGOTIATE message, whereas the origin-side machine sends a PADDING_NEGOTIATED message and between 7 to 10 DROP cells. This means that the average overhead of this machine is 11 padding cells per introduction circuit.

For the rend circuit case, this machine is quite light. Both sides send 2 padding cells, for a total of 4 padding cells.

Circuit padding consensus parameters

The circuit padding system has a handful of consensus parameters that can either disable circuit padding entirely, or rate limit the total overhead at relays and clients.

```
  * circpad_padding_disabled
    - If set to 1, no circuit padding machines will negotiate, and all
      current padding machines will cease padding immediately.
    - Default: 0

  * circpad_padding_reduced
    - If set to 1, only circuit padding machines marked as
"reduced"/"low
      overhead" will be used. (Currently no such machines are marked
      as "reduced overhead").
    - Default: 0

  * circpad_global_allowed_cells
    - This is the number of padding cells that must be sent before
      the 'circpad_global_max_padding_percent' parameter is applied.
    - Default: 0

  * circpad_global_max_padding_percent
    - This is the maximum ratio of padding cells to total cells,
specified
      as a percent. If the global ratio of padding cells to total
cells
      across all circuits exceeds this percent value, no more padding
is sent
      until the ratio becomes lower. 0 means no limit.
    - Default: 0

  * circpad_max_circ_queued_cells
    - This is the maximum number of cells that can be in the
circuitmux queue
      before padding stops being sent on that circuit.
    - Default: CIRCWINDOW_START_MAX (1000)
```

```
        "enc-key-cert" NL certificate NL

          [Exactly once per introduction point]

          Cross-certification of the encryption key using the
descriptor
          signing key.

          For "ntor" keys, certificate is a proposal 220 certificate
          wrapped in "-----BEGIN ED25519 CERT-----" armor.

          The subject
          key is the the ed25519 equivalent of a curve25519 public
          encryption key (`KP_hss_ntor`), with the ed25519 key
          derived using the process in proposal 228 appendix A,
          and its sign bit set to zero.

          The
          signing key is the descriptor signing key
(`KP_hs_desc_sign`).
          The certificate type must be [0B], and the signing-key
          extension is mandatory.

          NOTE: As with "auth-key", this certificate was intended to
be
          constructed the other way around.  However, for
compatibility
          with C tor, implementations need to construct it this way.
It
          serves even less point than "auth-key", however, since the
          encryption key `KP_hss_ntor` is already available from
          the `enc-key` entry.

          ALSO NOTE: Setting the sign bit of the subject key
          to zero makes the subjected unusable for verification;
          this is also a mistake preserved for compatiblility with
          C tor.

        "legacy-key" NL key NL

          [None or at most once per introduction point]
          [This field is obsolete and should never be generated; it
           is included for historical reasons only.]

          The key is an ASN.1 encoded RSA public key in PEM format
used for a
          legacy introduction point as described in
[LEGACY_EST_INTRO].

          This field is only present if the introduction point only
supports
          legacy protocol (v2) that is <= 0.2.9 or the subprotocol
value
          "HSIntro=3".
```

```
        "enc-key" SP KeyType SP key.. NL

          [Any number of times]

          Implementations should accept other types of onion keys
using this
          syntax (where "KeyType" is some string other than "ntor");
          unrecognized key types should be ignored.
```

Acknowledgments

This research was supported in part by NSF grants CNS-1111539, CNS-1314637, CNS-1526306, CNS-1619454, and CNS-1640548.

1. https://en.wikipedia.org/wiki/NetFlow
2. http://infodoc.alcatel-lucent.com/html/0_add-h-f/93-0073-10-01/7750_SR_OS_Router_Configuration_Guide/Cflowd-CLI.html
3. http://www.cisco.com/en/US/docs/ios/12_3t/netflow/command/reference/nfl_a1gt_ps5207_TSD_Products_Command_Reference_Chapter.html#wp1185203
4. http://www.cisco.com/c/en/us/support/docs/switches/catalyst-6500-series-switches/70974-netflow-catalyst6500.html#opconf
5. https://www.juniper.net/techpubs/software/erx/junose60/swconfig-routing-vol1/html/ip-jflow-stats-config4.html#560916
6. http://www.jnpr.net/techpubs/en_US/junos15.1/topics/reference/configuration-statement/flow-active-timeout-edit-forwarding-options-po.html
7. http://www.jnpr.net/techpubs/en_US/junos15.1/topics/reference/configuration-statement/flow-active-timeout-edit-forwarding-options-po.html
8. http://www.h3c.com/portal/Technical_Support___Documents/Technical_Documents/Switches/H3C_S9500_Series_Switches/Command/Command/H3C_S9500_CM-Release1648%5Bv1.24%5D-System_Volume/200901/624854_1285_0.htm#_Toc217704193
9. http://docs-legacy.fortinet.com/fgt/handbook/cli52_html/FortiOS%205.2%20CLI/config_system.23.046.html
10. http://wiki.mikrotik.com/wiki/Manual:IP/Traffic_Flow
11. https://metrics.torproject.org/dirbytes.html
12. http://freehaven.net/anonbib/cache/murdoch-pet2007.pdf
13. https://spec.torproject.org/proposals/188-bridge-guards.html
14. http://www.ntop.org/wp-content/uploads/2013/03/nProbe_UserGuide.pdf
15. http://arxiv.org/pdf/1512.00524
16. https://www.cs.kau.se/pulls/hot/thebasketcase-ape/
17. https://github.com/torproject/tor/tree/master/doc/HACKING/CircuitPaddingDevelopment.md
18. https://www.usenix.org/node/190967
    https://blog.torproject.org/technical-summary-usenix-fingerprinting-paper

Denial-of-service prevention mechanisms in Tor

This document covers the strategy, motivation, and implementation for denial-of-service mitigation systems designed into Tor.

The older `dos-spec` document is now the Memory exhaustion section here.

An in-depth description of the proof of work mechanism for onion services, originally proposal 327, is now in the Proof of Work for onion service introduction spec.

```
        "onion-key" SP "ntor" SP key NL

          [Exactly once per introduction point]

          The key is a base64 encoded curve25519 public key which is
the onion
          key of the introduction point Tor node used for the ntor
handshake
          when a client extends to it.


        "onion-key" SP KeyType SP key.. NL

          [Any number of times]

          Implementations should accept other types of onion keys
using this
          syntax (where "KeyType" is some string other than "ntor");
          unrecognized key types should be ignored.



        "auth-key" NL certificate NL

          [Exactly once per introduction point]

          The certificate is a proposal 220 certificate wrapped in
          "-----BEGIN ED25519 CERT-----".  It contains the
introduction
          point authentication key (`KP_hs_ipt_sid`), signed by
          the descriptor signing key (`KP_hs_desc_sign`).  The
          certificate type must be [09], and the signing key extension
          is mandatory.

          NOTE: This certificate was originally intended to be
          constructed the other way around: the signing and signed
keys
          are meant to be reversed.  However, C tor implemented it
          backwards, and other implementations now need to do the same
          in order to conform.  (Since this section is inside the
          descriptor, which is _already_ signed by `KP_hs_desc_sign`,
          the verification aspect of this certificate serves no point
in
          its current form.)


        "enc-key" SP "ntor" SP key NL

          [Exactly once per introduction point]

          The key is a base64 encoded curve25519 public key used to
encrypt
          the introduction request to service. (`KP_hss_ntor`)
```

```
connection
                  without PoW.
```

Followed by zero or more introduction points as follows (see section [NUM_INTRO_POINT] below for accepted values):

```
        "introduction-point" SP link-specifiers NL

          [Exactly once per introduction point at start of
introduction
            point section]

          The link-specifiers is a base64 encoding of a link specifier
          block in the format described in [BUILDING-BLOCKS] above.

          As of 0.4.1.1-alpha, services include both IPv4 and IPv6
link
          specifiers in descriptors. All available addresses SHOULD be
          included in the descriptor, regardless of the address that
the
          onion service actually used to connect/extend to the intro
          point.

          The client SHOULD NOT reject any LSTYPE fields which it
doesn't
          recognize; instead, it should use them verbatim in its
EXTEND
          request to the introduction point.

          The client SHOULD perform the basic validity checks on the
link
          specifiers in the descriptor, described in `tor-spec.txt`
          section 5.1.2. These checks SHOULD NOT leak
          detailed information about the client's version,
configuration,
          or consensus. (See 3.3 for service link specifier handling.)

          When connecting to the introduction point, the client SHOULD
send
          this list of link specifiers verbatim, in the same order as
given
          here.

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

Overview

As a public and anonymous network, Tor is open to many types of denial-of-service attempts. It's necessary to constantly develop a variety of defenses that mitigate specific types of attacks.

These mitigations are expected to improve network availability, but DoS mitigation is also important for limiting the avenues an attacker could use to perform active attacks on anonymity. For example, the ability to kill targeted Tor instances can be used to facilitate traffic analysis. See the "Sniper Attack" paper by Jansen, Tschorsch, Johnson, and Scheuermann.

The attack and defense environment changes over time. Expect that this document is an attempt to describe the current state of things, but that it may not be complete.

The defenses here are organized by the type of resource under contention. These can be physical resources (Memory, CPU, Bandwidth) or protocol resources (Channels, Circuits, Introductions).

In practice there are always overlaps between these resource types. Connecting to an onion service, for example, puts some strain on every resource type here.

Physical resources

Memory

Memory exhaustion is both one of the most serious denial-of-service avenues and the subject of the most fully developed defense mechanisms so far. We track overall memory use and free the most disposable objects first when usage is over threshold.

CPU

The available CPU time on a router can be exhausted, assuming the implementation is not capable of processing network input at line rate in all circumstances. This is especially problematic in the single-threaded C implementation. Certain expensive operations like circuit extension handshakes are deferred to a thread pool, but time on the main thread is still a precious resource.

We currently don't directly monitor and respond to CPU usage. Instead C Tor relies on limits for protocol resources, like circuits extensions and onion service introductions, that are associated with this CPU load.

Bandwidth

Relay operators can place hard limits on total bandwidth using the `Bandwidth` or `RelayBandwidth` options. These options can help relay operators avoid bandwidth peaks on their network, however they aren't designed as denial of service prevention mechanisms.

Beyond just shaving off harmful bandwidth peaks it's important that normal service is not disrupted too much, and especially not disrupted in a targetable way. To approximate this goal we rely on flow control and fair dequeueing of relay cells.

Protocol resources

Channels

All channels to some extent are a limited resource, but we focus specifically on preventing floods of incoming TLS connections.

Excessive incoming TLS connections consume memory as well as limited network and operating system resources. Excessive incoming connections typically signal a low-effort denial of service attack.

The C Tor implementation establishes limits on both the number of concurrent connections per IP address and the rate of new connections, using the `DoSConnection` family of configuration options and their corresponding consensus parameters.

Circuits

Excessive circuit creation can impact the entire path of that circuit, so it's important to reject these attacks any time they can be identified. Ideally we reject them as early as possible, before they have fully built the circuit.

Because of Tor's anonymity, most affected nodes experience the circuit flood as coming from every direction. The guard position, however, has a chance to notice specific peers that are creating too many circuits.

```
     "pow-params" SP scheme SP seed-b64 SP suggested-effort
                  SP expiration-time NL

      [At most once per scheme]

      If present, this line provides parameters for an optional proof-
of-work
      client puzzle. A client that supports an offered scheme can
include a
      corresponding solution in its introduction request to improve
priority
      in the service's processing queue.

      "pow-params" may appear multiple times; each occurrence must
have
      a different "scheme" field. Only the scheme `v1` is currently
defined.

      Unknown schemes found in a descriptor must be completely
ignored:
      future schemes might have a different format (in the parts of
the
      Item after the "scheme"; this could even include an Object); and
      future schemes might allow repetition, and might appear in any
order.
      Implementations SHOULD ignore lines with unrecognized schemes.

      Introduced in tor-0.4.8.1-alpha.

        scheme: The PoW system used. We call the one specified here
"v1".

        seed-b64: A random seed that should be used as the input to
the PoW
                  hash function. Should be 32 random bytes encoded in
base64
                  without trailing padding.

        suggested-effort: An unsigned integer specifying an effort
value that
                  clients should aim for when contacting the service.
Can be
                  zero to mean that PoW is available but not currently
                  suggested for a first connection attempt.

        expiration-time: A UTC timestamp in "YYYY-MM-DDTHH:MM:SS"
format (iso time
                  with no space) after which the above seed expires
and
                  is no longer valid as the input for PoW.

                  If a client encounters an expired seed,
                  it should either fetch a new descriptor,
                  or (if the descriptor is very recent) make a
```

```
    "intro-auth-required" SP types NL

     [At most once]

     A space-separated list of introduction-layer authentication
types; see
     section [INTRO-AUTH] for more info. A client that does not
support at
     least one of these authentication types will not be able to
contact the
     host. Recognized types are: 'ed25519'.


    "single-onion-service"

     [At most once]

     If present, this line indicates that the service is a Single
Onion
     Service (see prop260 for more details about that type of
service). This
     field has been introduced in 0.3.0 meaning 0.2.9 service don't
include
     this.
```

The C Tor implementation limits the acceptable rate of circuit creation per client IP address using the `DoSCircuit` configuration options and their corresponding consensus parameters.

Onion service introductions

Flooding an onion service with introduction attempts causes significant network load. In addition to the CPU, memory, and bandwidth load experienced by the introduction point and the service, all involved relays experience a circuit creation flood.

We have two types of onion service DoS mitigations currently. Both are optional, enabled as needed by individual onion servce operators.

Mitigation by rate limiting

Introduction attempts can be rate-limited by each introduction point, at the request of the service.

This defense is configured by an operator using the `HiddenServiceEnableIntroDos` configuration options. Services use the introduction DoS extension to communicate these settings to each introduction point.

Mitigation using proof of work

A short non-interactive computational puzzle can be solved with each connection attempt. Requests provided by the client will be entered into a queue prioritized by their puzzle solution's effort score. Requests are processed by the service at a limited rate, which can be adjusted to a value within the server's capabilities.

Based on the queue behavior, servers will continuously provide an updated effort suggestion. Queue backlogs cause the effort to rise, and an idle server will cause the effort to decay. If the queue is never overfull the effort decays to zero, asking clients not to include a proof-of-work solution at all.

We may support multiple cryptographic algorithms for this puzzle in the future, but currently we support one type. It's called `v1` in our protocol, and it's based on the Equi-X algorithm developed for this purpose. See the document on Proof of Work for onion service introduction.

This defense is configured by an operator using the `HiddenServicePoW` configuration options. Additionally, it requires both the client and the onion service to be compiled with the `pow` module (and `--enable-gpl` mode) available.

Despite this non-default build setting, proof of work *is* available through common packagers like the Tor Browser and Debian.

```
      SECRET_DATA = blinded-public-key | descriptor_cookie
      STRING_CONSTANT = "hsdir-encrypted-data"

   If restricted discovery is disabled the 'descriptor_cookie' field
is left blank.

   The ciphertext is placed on the "encrypted" field of the
descriptor.
```

Second layer plaintext format

After decrypting the second layer ciphertext, clients can finally learn the list of intro points etc. The plaintext has the following format:

`proto` — Declare supported subprotocol capabilities

- `proto` *entry entry* ..
- At most once
- Syntax and semantics as for `proto` in a server descriptor.

This entry declares that the onion service supports one or more subprotocol capabilities.

Not all subprotocol capabilities should be listed here: only the ones whose presence the client may need to know in advance, as listed in the table below:

Name	Capability
RELAY_NEGOTIATE_SUBPROTO	“Relay=5”
RELAY_CRYPTO_CGO	“Relay=6”

(unconverted items in older spec format)

```
"create2-formats" SP formats NL

\[Exactly once\]

      A space-separated list of integers denoting CREATE2 cell HTYPEs
      (handshake types) that the server recognizes. Must include at
least
      ntor as described in tor-spec.txt. See tor-spec section 5.1 for
a list
      of recognized handshake types.
```

Hiding restricted discovery data

```
    Hidden services should avoid leaking whether restricted discovery
is
    enabled or how many authorized clients there are.

    Hence even when restricted discovery is disabled, the hidden
service adds
    fake "desc-auth-type", "desc-auth-ephemeral-key" and "auth-client"
lines to
    the descriptor, as described in [HS-DESC-FIRST-LAYER].

    The hidden service also avoids leaking the number of authorized
clients by
    adding fake "auth-client" entries to its descriptor. Specifically,
    descriptors always contain a number of authorized clients that is
a
    multiple of 16 by adding fake "auth-client" entries if needed.
    [XXX consider randomization of the value 16]

    Clients MUST accept descriptors with any number of "auth-client"
lines as
    long as the total descriptor size is within the max limit of 50k
(also
    controlled with a consensus parameter).
```

Second layer of encryption

The second layer of descriptor encryption is designed to protect descriptor confidentiality against unauthorized clients. If restricted discovery is enabled, it's encrypted using the descriptor_cookie, and contains needed information for connecting to the hidden service, like the list of its introduction points.

If restricted discovery is disabled, then the second layer of HS encryption does not offer any additional security, but is still used.

Second layer encryption keys

The encryption keys and format for the second layer of encryption are generated as specified in [HS-DESC-ENCRYPTION-KEYS] with customization parameters as follows:

Memory exhaustion

Memory exhaustion is a broad issue with many underlying causes. The Tor protocol requires clients, onion services, relays, and authorities to store various kind of information in buffers and caches. But an attacker can use these buffers and queues to exhaust the memory of the a targeted Tor process, and force the operating system to kill that process.

With this in mind, any Tor implementation (especially one that runs as a relay or onion service) must take steps to prevent memory-based denial-of-service attacks.

Detecting low memory

The easiest way to notice you're out of memory would, in theory, be getting an error when you try to allocate more. Unfortunately, some systems (e.g. Linux) won't actually give you an "out of memory" error when you're low on memory. Instead, they overcommit and promise you memory that they can't actually provide… and then later on, they might kill processes that actually try to use more memory than they wish they'd given out.

So in practice, the mainline Tor implementation uses a different strategy. It uses a self-imposed "MaxMemInQueues" value as an upper bound for how much memory it's willing to allocate to certain kinds of queued usages. This value can either be set by the user, or derived from a fraction of the total amount of system RAM.

As of Tor 0.4.7.x, the MaxMemInQueues mechanism tracks the following kinds of allocation:

- Relay cells queued on circuits.
- Per-connection read or write buffers.
- On-the-fly compression or decompression state.
- Half-open stream records.
- Cached onion service descriptors (hsdir only).
- Cached DNS resolves (relay only).
- GEOIP-based usage activity statistics.

Note that directory caches aren't counted, since those are stored on disk and accessed via mmap.

Responding to low memory

If our allocations exceed MaxMemInQueues, then we take the following steps to reduce our memory allocation.

Freeing from caches: For each of our onion service descriptor cache, our DNS cache, and our GEOIP statistics cache, we check whether they account for greater than 20% of our total allocation. If they do, we free memory from the offending cache until the total remaining is no more than 10% of our total allocation.

When freeing entries from a cache, we aim to free (approximately) the oldest entries first.

Freeing from buffers: After freeing data from caches, we see whether allocations are still above 90% of MaxMemInQueues. If they are, we try to close circuits and connections until we are below 90% of MaxMemInQueues.

When deciding to what circuits to free, we sort them based on the age of the oldest data in their queues, and free the ones with the oldest data. (For example, a circuit on which a single cell has been queued for 5 minutes would be freed before a circuit where 100 cells have been queued for 5 seconds.) “Data queued on a circuit” includes all data that we could drop if the circuit were destroyed: not only the cells on the circuit’s relay cell queue, but also any bytes queued in buffers associated with streams or half-stream records attached to the circuit.

We free non-tunneled directory connections according to a similar rule, according to the age of their oldest queued data.

Upon freeing a circuit, a “DESTROY cell” must be sent in both directions.

Reporting low memory

We define a “low threshold” equal to 3/4 of MaxMemInQueues. Every time our memory usage is above the low threshold, we record ourselves as being “under memory pressure”.

(This is not currently reported.)

Client behavior

```
    The goal of clients at this stage is to decrypt the "encrypted"
field as
    described in [HS-DESC-SECOND-LAYER].

    If restricted discovery is enabled, authorized clients need to
extract the
    descriptor cookie to proceed with decryption of the second layer
as
    follows:

    An authorized client parsing the first layer of an encrypted
descriptor,
    extracts the ephemeral key from "desc-auth-ephemeral-key" and
calculates
    CLIENT-ID and COOKIE-KEY as described in the section above using
their
    x25519 private key. The client then uses CLIENT-ID to find the
right
    "auth-client" field which contains the ciphertext of the
descriptor
    cookie. The client then uses COOKIE-KEY and the iv to decrypt the
    descriptor_cookie, which is used to decrypt the second layer of
descriptor
    encryption as described in [HS-DESC-SECOND-LAYER].
```

```
      - The "client-id" field is CLIENT-ID from above encoded in
base64.

      - The "iv" field is 16 random bytes encoded in base64.

      - The "encrypted-cookie" field contains the descriptor cookie
ciphertext
        as follows and is encoded in base64:
           encrypted-cookie = STREAM(iv, COOKIE-KEY) XOR
N_hs_desc_enc.

      See section [FIRST-LAYER-CLIENT-BEHAVIOR] for the client-side
logic of
      how to decrypt the descriptor cookie.

    "encrypted" NL encrypted-string

     [Exactly once]

      An encrypted blob containing the second layer ciphertext, whose
format is
      discussed in [HS-DESC-SECOND-LAYER] below. The blob is base64
encoded
      and enclosed in -----BEGIN MESSAGE---- and ----END MESSAGE----
wrappers.

    Compatibility note: The C Tor implementation does not include a
final
    newline when generating this first-layer-plaintext section; other
    implementations MUST accept this section even if it is missing its
final
    newline.  Other implementations MAY generate this section without
a final
    newline themselves, to avoid being distinguishable from C tor.
```

Tor's extensions to the SOCKS protocol

Overview

The SOCKS protocol provides a generic interface for TCP proxies. Client software connects to a SOCKS server via TCP, and requests a TCP connection to another address and port. The SOCKS server establishes the connection, and reports success or failure to the client. After the connection has been established, the client application uses the TCP stream as usual.

Except as noted below, Tor supports SOCKS4 as defined here, SOCKS4A as defined here, and SOCKS5 as defined in RFC 1928 and RFC 1929.

The stickiest issue for Tor in supporting clients, in practice, is forcing DNS lookups to occur at the OR side: if clients do their own DNS lookup, the DNS server can learn which addresses the client wants to reach. SOCKS4 supports addressing by IPv4 address; SOCKS4A is a kludge on top of SOCKS4 to allow addressing by hostname; SOCKS5 supports IPv4, IPv6, and hostnames.

Extent of support

Tor supports the SOCKS4, SOCKS4A, and SOCKS5 standards, *except as follows*:

SOCKS4, **SOCKS4A**:

- The BIND command is not supported.

SOCKS5:

- The SOCKS5 "UDP ASSOCIATE" command is not supported.
- The SOCKS5 "BIND" command is not supported.
- SOCKS5 GSSAPI authentication, and its subnegotiation protocol, are not supported, even though they are listed as "MUST support" by RFC 1928.
- As an extension to support some broken clients, the C tor implementation allows clients to pass "USERNAME/PASSWORD" authentication message to us even if "NO AUTHENTICATION REQUIRED" was selected in the "METHOD selection message". (In such cases, the provided "USERNAME/PASSWORD" authentication message is interpreted as if "USERNAME/PASSWORD"

authentication had been selected in the “METHOD selection message.”) This technically violates RFC 1929, but ensures interoperability with somewhat broken SOCKS5 client implementations.

Name lookup

As an extension to SOCKS4A and SOCKS5, Tor implements a new command value, “RESOLVE” ([F0]). When Tor receives a RESOLVE SOCKS command, it initiates a remote lookup of the hostname provided as the target address in the SOCKS request. The reply is either an error (if the address couldn’t be resolved) or a success response. In the case of success, the address is stored in the portion of the SOCKS response reserved for remote IP address.

(We support RESOLVE in SOCKS4 too, even though it is unnecessary.)

For SOCKS5 only, we support reverse resolution with a new command value, RESOLVE_PTR ([F1]). In response to a RESOLVE_PTR SOCKS5 command with an IPv4 address as its target, Tor attempts to find the canonical hostname for that IPv4 record, and returns it in the “server bound address” portion of the reply. (This command was not supported before Tor 0.1.2.2-alpha.)

HTTP-resistance

Tor checks the first byte of each SOCKS request to see whether it looks more like an HTTP request (that is, it starts with a “G”, “H”, or “P”). If so, Tor returns a small webpage, telling the user that their browser is misconfigured. This is helpful for the many users who mistakenly try to use Tor as an HTTP proxy instead of a SOCKS proxy.

Optimistic data

Tor allows SOCKS clients to send connection data before Tor has sent a SOCKS response. Tor will package such data and send it to the exit, without waiting to see whether the connection attempt succeeds. This behavior can save a single round-trip time when starting connections with a protocol where the client speaks first (like HTTP). Clients that do this must be ready to hear that their connection has succeeded or failed *after* they have sent the data.

```
      This field contains the type of authorization used to protect
the
      descriptor. The only recognized type is "x25519" and specifies
the
      encryption scheme described in this section.

      If restricted discovery is disabled, the value here should be
"x25519".

     "desc-auth-ephemeral-key" SP KP_hs_desc_ephem NL

      [Exactly once]

      This field contains `KP_hss_desc_enc`, an ephemeral x25519
public
      key generated by the hidden service and encoded in base64. The
key
      is used by the encryption scheme below.

      If restricted discovery is disabled, the value here should be a
fresh
      x25519 pubkey that will remain unused.

     "auth-client" SP client-id SP iv SP encrypted-cookie

      [At least once]

      When restricted discovery is enabled, the hidden service inserts
an
      "auth-client" line for each of its authorized clients. If client
      authorization is disabled, the fields here can be populated with
random
      data of the right size (that's 8 bytes for 'client-id', 16 bytes
for 'iv'
      and 16 bytes for 'encrypted-cookie' all encoded with base64).

      When restricted discovery is enabled, each "auth-client" line
      contains the descriptor cookie `N_hs_desc_enc` encrypted to each
      individual client. We assume that each authorized client
possesses
      a pre-shared x25519 keypair (`KP_hsc_desc_enc`) which is used to
      decrypt the descriptor cookie.

      We now describe the descriptor cookie encryption scheme. Here is
what
      the hidden service computes:

          SECRET_SEED = x25519(KS_hs_desc_ephem, KP_hsc_desc_enc)
          KEYS = SHAKE256_KDF(N_hs_subcred | SECRET_SEED, 40)
          CLIENT-ID = fist 8 bytes of KEYS
          COOKIE-KEY = last 32 bytes of KEYS

      Here is a description of the fields in the "auth-client" line:
```

First layer plaintext format

After clients decrypt the first layer of encryption, they need to parse the plaintext to get to the second layer ciphertext which is contained in the “encrypted” field.

If restricted discovery is enabled, the hidden service generates a fresh descriptor_cookie key (`N_hs_desc_enc`, 32 random bytes) and encrypts it using each authorized client’s identity x25519 key. Authorized clients can use the descriptor cookie (`N_hs_desc_enc`) to decrypt the second (inner) layer of encryption. Our encryption scheme requires the hidden service to also generate an ephemeral x25519 keypair for each new descriptor.

If restricted discovery is disabled, fake data is placed in each of the fields below to obfuscate whether restricted discovery is enabled.

Here are all the supported fields:

“desc-auth-type” SP type NL

[Exactly once]

Extended error codes

We define a set of additional extension error codes that can be returned by our SOCKS implementation in response to failed onion service connections.

(In the C Tor implementation, these error codes can be disabled via the ExtendedErrors flag. In Arti, these error codes are enabled whenever onion services are.)

- `[F0]` Onion Service Descriptor Can Not be Found

  The requested onion service descriptor can’t be found on the hashring and thus not reachable by the client.

- `[F1]` Onion Service Descriptor Is Invalid

  The requested onion service descriptor can’t be parsed or signature validation failed.

- `[F2]` Onion Service Introduction Failed

  Client failed to introduce to the service meaning the descriptor was found but the service is not anymore at the introduction points. The service has likely changed its descriptor or is not running.

- `[F3]` Onion Service Rendezvous Failed

  Client failed to rendezvous with the service which means that the client is unable to finalize the connection.

- `[F4]` Onion Service Missing Client Authorization

  Tor was able to download the requested onion service descriptor but is unable to decrypt its content because it is missing client authorization information for it.

- `[F5]` Onion Service Wrong Client Authorization

  Tor was able to download the requested onion service descriptor but is unable to decrypt its content using the client authorization information it has. This means the client access were revoked.

- `[F6]` Onion Service Invalid Address

The given .onion address is invalid. In one of these cases this error is returned: address checksum doesn't match, ed25519 public key is invalid or the encoding is invalid.

- `[F7]` Onion Service Introduction Timed Out

  Similar to `[F2]` code but in this case, all introduction attempts have failed due to a time out.

(Note that not all of the above error codes are currently returned by Arti as of August 2023.)

Interpreting usernames and passwords

We support a series of extensions in SOCKS5 Username/Passwords. Currently, these extensions can encode a stream isolation parameter (used to indicate that streams may share a circuit) and an RPC object ID (used to associate the stream with an entity in an RPC session).

These extensions are in use whenever the SOCKS5 Username begins with the 8-byte "magic" sequence `[3c 74 6f 72 53 30 58 3e]`. (This is the ASCII encoding of `<torS0X>`).

If the SOCKS5 Username/Password fields are present but the Username does not begin with this byte sequence, it indicates *legacy isolation*. New client implementations SHOULD NOT use legacy isolation. A SocksPort may be configured to reject legacy isolation.

When these extensions are in use, the next byte of the username after the "magic" sequence indicate a format type. Any implementation receiving an unrecognized or missing format type MUST reject the socks request.

- When the format type is `[30]` (the ascii encoding of `0`), we interpret the rest of the Username field and the Password field as follows:

  The remainder of the Username field must be empty.

  The Password field is stream isolation parameter. If it is empty, the stream isolation parameter is an empty string.

- When the format type is `[31]` (the ascii encoding of `1`), we interpret the rest of the Username and field and the Password field as follows:

Hidden service descriptors: encryption format

Hidden service descriptors are protected by two layers of encryption. Clients need to decrypt both layers to connect to the hidden service.

The first layer of encryption provides confidentiality against entities who don't know the public key of the hidden service (e.g. HSDirs), while the second layer of encryption is only useful when restricted discovery is enabled and protects against entities that do not possess valid client credentials.

First layer of encryption

The first layer of HS descriptor encryption is designed to protect descriptor confidentiality against entities who don't know the public identity key of the hidden service.

First layer encryption logic

The encryption keys and format for the first layer of encryption are generated as specified in [HS-DESC-ENCRYPTION-KEYS] with customization parameters:

```
SECRET_DATA = blinded-public-key
STRING_CONSTANT = "hsdir-superencrypted-data"
```

The encryption scheme in [HS-DESC-ENCRYPTION-KEYS] uses the service credential which is derived from the public identity key (see [SUBCRED]) to ensure that only entities who know the public identity key can decrypt the first descriptor layer.

The ciphertext is placed on the "superencrypted" field of the descriptor.

Before encryption the plaintext is padded with NUL bytes to the nearest multiple of 10k bytes.

```
     "revision-counter" SP Integer NL

       [Exactly once.]

       The revision number of the descriptor. If an HSDir receives a
       second descriptor for a key that it already has a descriptor
for,
       it should retain and serve the descriptor with the higher
       revision-counter.

       (Checking for monotonically increasing revision-counter values
       prevents an attacker from replacing a newer descriptor signed
by
       a given key with a copy of an older version.)

       Implementations MUST be able to parse 64-bit values for these
       counters.


     "superencrypted" NL encrypted-string

       [Exactly once.]

       An encrypted blob, whose format is discussed in [HS-DESC-ENC]
below. The
       blob is base64 encoded and enclosed in -----BEGIN MESSAGE----
and
       ----END MESSAGE---- wrappers. (The resulting document does not
end with
       a newline character.)


     "signature" SP signature NL

       [exactly once, at end.]

       A signature of all previous fields, using the signing key in
the
       descriptor-signing-key-cert line, prefixed by the string "Tor
onion
       service descriptor sig v3". We use a separate key for signing,
so that
       the hidden service host does not need to have its private
blinded key
       online.
```

HSDirs accept hidden service descriptors of up to 50k bytes (a consensus parameter should also be introduced to control this value).

The remainder of the Username field encodes an RPC Object ID. It must not be empty.

The Password field is stream isolation parameter. If it is empty, the stream isolation parameter is an empty string.

All implementations SHOULD implement format type `[30]`.

Tor began supporting format type `[30]` in 0.4.9.1-alpha. Arti began supporting format types `[30]` and `[31]` in 1.2.8.

Note however that using format type `[30]` will have the intended effect with older versions of Tor and Arti, even though they will interpret it as legacy isolation.

Examples:

- Username=`hello`; Password=`world`. These are legacy parameters, since `hello` does not begin with `<torS0X>`.
- Username=`<torS0X>0`; Password=`123`. There is no associated object ID. The isolation string is `123`.
- Username=`<torS0X>0`; Password is empty. There is no associated object ID. The isolation string is empty.
- Username=`<torS0X>1abc`; Password=`123`. The object ID is `abc`. The isolation string is `123`.
- Username=`<torS0X>1abc`; Password is empty. The object ID is `abc`. The isolation string is empty.
- Username=`<torS0X>0abc`; Password=`123`. Error: The format type is `0` but there is extra data in the username. Implementations must reject this.
- Username=`<torS0X>1`; Password=`123`. Error: The format type is `1` but the object ID is empty. Implementations must reject this.
- Username=`<torS0X>9`; Password=`123`. Error: The format type `9` is unspecified. Implementations must reject this.
- Username=`<torS0X>`; Password=`123`. Error: There is no format type. Implementations must reject this.

Stream isolation

Two streams are considered to have the same SOCKS authentication values if and only if one of the following is true:

- They are both SOCKS4 or SOCKS4a, with the same user "ID" string.
- They are both SOCKS5, with no authentication.
- They are both SOCKS5 with USERNAME/PASSWORD authentication, using legacy isolation parameters, and they have identical usernames and identical passwords.
- They are both SOCKS5 using the extensions above, with the same stream isolation parameter.

Additionally, two streams with different format types (e.g. `[30]` and `[31]`) may not share a circuit.

> For more information on stream isolation, including other factors that can prevent streams from sharing circuits, see proposal 171.

Inferring IP version preference

The Tor protocol's `BEGIN` messages include flags to indicate preferences for IPv4 versus IPv6.

When opening streams based on SOCKS requests, the "IPv6 okay" flag is set, and the other flags are left cleared, except as follows:

If the address is an IPv4 address (received via SOCKS4, SOCKS5 address type 1, or a hostname string containing a literal IPv4 address) then the "IPv6 okay" flag is cleared.

If the address is an IPv6 address (via SOCKS5 address type 4, or a hostname string containing a literal IPv6 address) then the "IPv4 not okay" flag is set.

User-specified configuration options may override this behavior.

Hidden service descriptors: outer wrapper [DESC-OUTER]

The format for a hidden service descriptor is as follows, using the meta-format from dir-spec.txt.

"hs-descriptor" SP version-number NL

[At start, exactly once.]

```
       The version-number is a 32 bit unsigned integer indicating the
version
       of the descriptor. Current version is "3".

     "descriptor-lifetime" SP LifetimeMinutes NL

       [Exactly once]

       The lifetime of a descriptor in minutes. An HSDir SHOULD expire
the
       hidden service descriptor at least LifetimeMinutes after it was
       uploaded.

       The LifetimeMinutes field can take values between 30 and 720
(12
       hours).



    "descriptor-signing-key-cert" NL certificate NL

       [Exactly once.]

       The 'certificate' field contains a certificate in the format
from
       proposal 220, wrapped with "-----BEGIN ED25519 CERT-----".  The
       certificate cross-certifies the short-term descriptor signing
key with
       the blinded public key.  The certificate type must be [08], and
the
       blinded public key must be present as the signing-key
extension.
```

To avoid client reachability issues in this rare event, hidden services should use the new shared random values to find the new responsible HSDirs and upload their descriptors there.

XXX How long should they upload descriptors there for?

Tor's support for HTTP CONNECT, and extensions

The C Tor implementation supports the HTTP CONNECT protocol for providing a TCP tunnel over the Tor network.

HTTP is more readily extensible than SOCKS, and so it allows applications and Tor proxies to better communicate structured information about their needs.

Here we will outline the amount of HTTP support that Tor provides, and describe the extensions that it supports.

HTTP CONNECT support

A Tor proxy supporting HTTP CONNECT *should* implement the CONNECT method as specified in RFC 9110 s9.3.6. It SHOULD NOT support any other HTTP methods.

HTTP 1.0 support is mandatory; other HTTP versions are optional.

> HTTP 1.0 doesn't actually specify the CONNECT method. But C Tor approximates an implementation of HTTP 1.0 only, so we do not currently require HTTP 1.1.

As with other proxy ports, Tor SHOULD NOT listen on publicly reachable addresses.

All CONNECT requests received at the HTTP CONNECT port should either be rejected, or handled by making a connection over the Tor network and forwarding their data. No request should be forwarded in any other way.

OPTIONS method support

Implementations SHOULD implement the OPTIONS method. It SHOULD return an `Allow` header indicating the permitted methods, which SHOULD be CONNECT and OPTIONS.

> Clients can use this feature to inspect `Server` and `Tor-Capabilities` headers without actually making a connection.

> When supported by the proxy, clients SHOULD use HTTP/1.1 to make OPTIONS requests, and use the same TCP connection for any CONNECT request that relies on its output. Using separate TCP connections for OPTIONS and CONNECT can result in time-of-check/time-of-use issues.

(Supported by Arti, and by C Tor since 0.4.9.4-alpha.)

Avoiding cross-site probing attacks

To prevent attackers from using DNS-rebinding to circumvent cross-site enforcement, the proxy MUST inspect the Host header on every request whose method is not CONNECT. If the target address is anything other than “localhost”, “127.0.0.1”, or “::1”, then the proxy MUST reject the request and close the stream **without replying, even with an error.**

> For defense in depth, Tor Browser already uses NoScript LAN protection to prevent connections to localhost altogether. This mechanism is still valuable, since it does help prevent a non-patched browser from being used to probe whether a Tor proxy is running at some known port.

(Arti and C Tor.)

Error codes

Tor provides the following mapping from Tor END reasons to HTTP error codes:

> These aren’t necessarily sensible, and will likely change in the future.

END reason	HTTP status code
MISC	500
RESOLVEFAILED	503 [1]
NOROUTE	503 [1]
CONNECTREFUSED	403
EXITPOLICY	503
DESTROY	502
DONE[2]	502
TIMEOUT	504
HIBERNATING	502
INTERNAL	502

Publishing shared random values

Our design for limiting the predictability of HSDir upload locations relies on a shared random value (SRV) that isn’t predictable in advance or too influenceable by an attacker. The authorities must run a protocol to generate such a value at least once per hsdir period. Here we describe how they publish these values; the procedure they use to generate them can change independently of the rest of this specification. For more information see [SHAREDRANDOM-REFS].

According to proposal 250, we add two new lines in consensuses:

```
"shared-rand-previous-value" SP NUM_REVEALS SP VALUE NL
"shared-rand-current-value" SP NUM_REVEALS SP VALUE NL
```

Client behavior in the absence of shared random values

If the previous or current shared random value cannot be found in a consensus, then Tor clients and services need to generate their own random value for use when choosing HSDirs.

To do so, Tor clients and services use:

```
SRV = SHA3_256("shared-random-disaster" | INT_8(period_length) |
INT_8(period_num))
```

where period_length is the length of a time period in minutes, rounded down; period_num is calculated as specified in [TIME-PERIODS] for the wanted shared random value that could not be found originally.

Hidden services and changing shared random values

It’s theoretically possible that the consensus shared random values will change or disappear in the middle of a time period because of directory authorities dropping offline or misbehaving.

These requests must be made anonymously, on circuits not used for anything else.

The legacy ".z" suffix is **not** supported for these URLs.

To avoid fingerprinting, clients SHOULD NOT advertise any compression methods in their `Accept-Encoding` headers for these requests that are not listed as must-implement for clients in the directory specification.

Client-side validation of onion addresses

When a Tor client receives a prop224 onion address from the user, it MUST first validate the onion address before attempting to connect or fetch its descriptor. If the validation fails, the client MUST refuse to connect.

As part of the address validation, Tor clients should check that the underlying ed25519 key does not have a torsion component. If Tor accepted ed25519 keys with torsion components, attackers could create multiple equivalent onion addresses for a single ed25519 key, which would map to the same service. We want to avoid that because it could lead to phishing attacks and surprising behaviors (e.g. imagine a browser plugin that blocks onion addresses, but could be bypassed using an equivalent onion address with a torsion component).

The right way for clients to detect such fraudulent addresses (which should only occur malevolently and never naturally) is to extract the ed25519 public key from the onion address and multiply it by the ed25519 group order and ensure that the result is the ed25519 identity element. For more details, please see [TORSION-REFS].

END reason	HTTP status code
RESOURCELIMIT	502
CONNRESET	503
TORPROTOCOL	502
NOTADIRECTORY[3]	500
anything else	500

Implementations SHOULD include information about the actual error code in the Reason-Phrase part of the Status-Line.

Note that once a connection has *succeeded*, no subsequent status code for an END reason will be sent, since Tor already sent a "200 OK".

> Note that in accordance with RFC 2119 s9.3.6, even after an unclean END, data sent by the peer (before END) MUST be delivered to the application if possible.

> TODO: Specify specific return values, and/or messages, and/or headers, for onion-service-specific issues.

Standard headers

> Except as noted here, and in "Extended headers" below, the C tor implementation doesn't parse or send any HTTP headers.
>
> If this is non-conformant, we should change it.

Proxy-Authorization: Backward compatible isolation

(Request only.)

The `Proxy-Authorization` header is sent by the application, and used as an input for stream isolation; see "Stream isolation" below.

Applications intending to use this behavior SHOULD use `basic` authentication, and set the username to `tor-iso`. Implementations SHOULD warn if the `Proxy-Authorization` header has a different value.

> C Tor does not impose an upper length limit on this header.

`Via` / `Server`: Reporting proxy software version

(Response only.)

An indication of which software has answered the request.

“Via” is appropriate in response to CONNECT requests; “Server” is appropriate in response to OPTIONS requests.

The “via” header should include the name of the software, and its version, as the comment field. In “via”, the received-protocol and received-by fields should be set to “tor/1.0” and “tor-network” respectively, as in:

```
Via: tor/1.0 tor-network (Tor 0.4.9.4-alpha)
```

In “Server”, we use “tor/1.0” as the product, and place the software name and version in its comment field, as in:

```
Server: tor/1.0 (Arti 1.7.0)
```

Proxies MUST NOT include “tor/1.0” here unless they forward user traffic over anonymous Tor connections.

Clients MAY check for the presence or absence of this header.

Clients MUST NOT inspect the contents of this header to determine whether a given feature is supported or not; they should use `Tor-Capabilities` instead.

Clients MAY use this to determine whether some software has a particular bug, but the matching SHOULD NOT treat any future versions as buggy.

> That is, the ideal approach is to use
>
> ```
> if (version in FIRST_BROKEN_VERSION .. LAST_BROKEN_VERSION) {
>    use workaround
> }
> ```
>
> but until the bug is fixed, developers will not know what `LAST_BROKEN_VERSION` is.
>
> What developers SHOULD NOT do is write `if (version >= FIRST_BROKEN_VERSION)`, since that will still apply the workaround when the bug is fixed.

Directory behavior for handling descriptor uploads [DIRUPLOAD]

Upon receiving a hidden service descriptor publish request, directories MUST check the following:

```
     * The outer wrapper of the descriptor can be parsed according to
       [DESC-OUTER]
     * The version-number of the descriptor is "3"
     * If the directory has already cached a descriptor for this
hidden service,
       the revision-counter of the uploaded descriptor must be greater
than the
       revision-counter of the cached one
     * The descriptor signature is valid
```

If any of these basic validity checks fails, the directory MUST reject the descriptor upload.

NOTE: Even if the descriptor passes the checks above, its first and second layers could still be invalid: directories cannot validate the encrypted layers of the descriptor, as they do not have access to the public key of the service (required for decrypting the first layer of encryption), or the necessary client credentials (for decrypting the second layer).

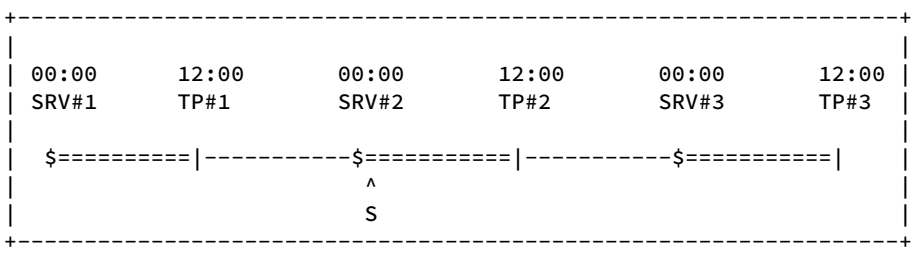
Expiring hidden service descriptors

Hidden services set their descriptor’s “descriptor-lifetime” field to 180 minutes (3 hours). Hidden services ensure that their descriptor will remain valid in the HSDir caches, by republishing their descriptors periodically as specified in [WHEN-HSDESC].

Hidden services MUST also keep their introduction circuits alive for as long as descriptors including those intro points are valid (even if that’s after the time period has changed).

URLs for anonymous uploading and downloading

Hidden service descriptors conforming to this specification are uploaded with an HTTP POST request to the URL `/tor/hs/<version>/publish` relative to the hidden service directory’s root, and downloaded with an HTTP GET request for the URL `/tor/hs/<version>/<z>` where `<z>` is a base64 encoding of the hidden service’s blinded public key and `<version>` is the protocol version which is “3” in this case.



Consider that the service is at 01:00 right after SRV#2: it will upload its first descriptor using TP#1 and SRV#1.

Second descriptor upload logic

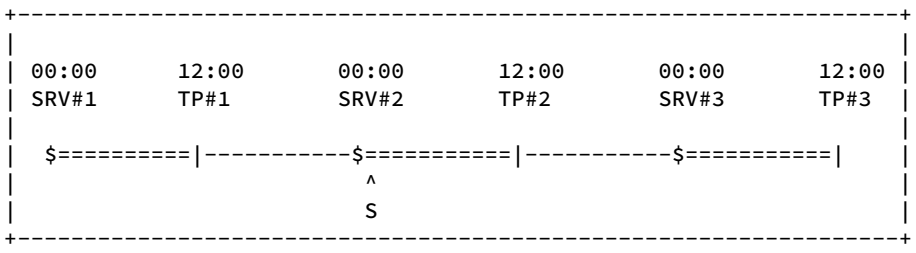
Here is the service logic for uploading its second descriptor:

When a service is in the time segment between a new time period a new SRV (i.e. the segments drawn with "-"), it uses the current time period and current SRV for uploading its second descriptor: that's meant to cover for clients that have an up-to-date consensus on the same TP as the service.

Example: Consider in the above illustration that the service is at 13:00 right after TP#1: it will upload its second descriptor using TP#1 and SRV#1.

Now if a service is in the time segment between a new SRV and a new time period (i.e. the segments drawn with "=") it uses the next time period and the current SRV for its second descriptor: that's meant to cover clients with a newer consensus than the service (in the next time period).

Example:



Consider that the service is at 01:00 right after SRV#2: it will upload its second descriptor using TP#2 and SRV#2.

(Supported by Arti, and by C Tor since 0.4.9.4-alpha.)

Extensions

Here we describe Tor's behaviors that are in addition to those of a standard HTTP CONNECT tunnel.

Determining IP version preference

Tor applies the same rules to HTTP CONNECT requests as it does to SOCKS5 requests when determining which IP versions to request or allow.

Extended headers

All new headers specified for use with Tor will begin with `Tor-`.

The "X-Tor-" prefix is reserved for historical extensions, and is deprecated for the reasons explained in RFC 6648.

Tor-Stream-Isolation: Stream isolation

(Request only.)

The `X-Tor-Stream-Isolation` header is sent by the application, and used as an input for stream isolation; see "Stream isolation" below.

It can appear only once; subsequent occurrences are ignored.

> C Tor has accepted this header since 0.4.9.4-alpha.
>
> C Tor does not impose an upper length limit on this header.

X-Tor-Stream-Isolation: Legacy stream isolation

(Request only.)

This is an old alternative to for `Tor-Stream-Isolation` for use with Tor 0.4.9.3-alpha and earlier.

All implementations SHOULD accept it, for backward compatibility.

> C Tor does not impose an upper length limit on this header.

`Tor-RPC-Target`: Arti RPC Support

(Request only.)

Contains an Arti RPC Object ID, to be used as a target for an RPC request.

Implementations SHOULD reject such requests if the target RPC object does not exist.

An implementation that generally supports Tor extensions, but that doesn't support Arti RPC, SHOULD reject requests containing this header.

> (Arti implements RPC behavior; C Tor implements "reject" behavior.)

`Tor-Request-Failed`: Extended error codes

(Response only.)

A space-separated list of tokens, each of which indicates a reason why the requested connection could not be completed.

> The specific members of this list are not currently documented.
>
> We intend that they will eventually correspond to all the current END reasons, and all the onion-service-specific SOCKS failure codes.

(Arti only.)

`Tor-Family-Preference`: IP version preferences

(Request only.)

Tor allows the client to transmit its preferences for IP versions as part of its BEGIN message.

We allow these to be encoded in an HTTP CONNECT header, `Tor-Family-Preference`.

Permitted values are:

- `ipv4-preferred` (any family allowed; ipv4 preferred)
- `ipv6-preferred` (any family allowed; ipv6 preferred)
- `ipv4-only` (only ipv4 is allowed)
- `ipv6-only` (only ipv6 is allowed)

If a client (C1) is at 13:00 right after TP#1, then it will use TP#1 and SRV#1 for fetching descriptors. Also, if a client (C2) is at 01:00 right after SRV#2, it will still use TP#1 and SRV#1.

Service behavior for uploading descriptors

As discussed above, services maintain two active descriptors at any time. We call these the "first" and "second" service descriptors. Services rotate their descriptor every time they receive a consensus with a valid_after time past the next SRV calculation time. They rotate their descriptors by discarding their first descriptor, pushing the second descriptor to the first, and rebuilding their second descriptor with the latest data.

Services like clients also employ a different logic for picking SRV and TP values based on their position in the graph above. Here is the logic:

First descriptor upload logic

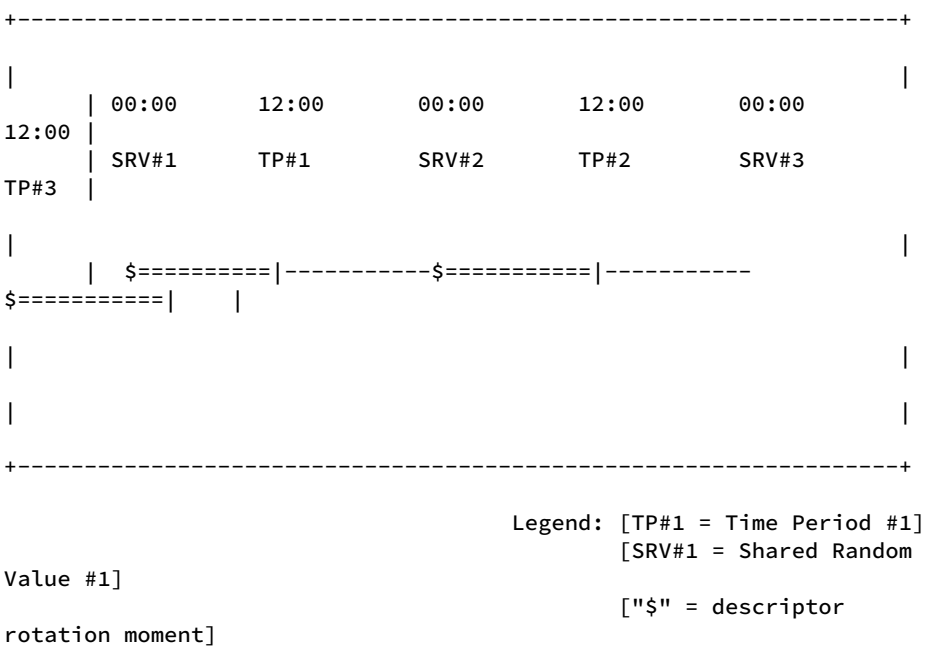
Here is the service logic for uploading its first descriptor:

When a service is in the time segment between a new time period a new SRV (i.e. the segments drawn with "-"), it uses the previous time period and previous SRV for uploading its first descriptor: that's meant to cover for clients that have a consensus that is still in the previous time period.

Example: Consider in the above illustration that the service is at 13:00 right after TP#1. It will upload its first descriptor using TP#0 and SRV#0. So if a client still has a 11:00 consensus it will be able to access it based on the client logic above.

Now if a service is in the time segment between a new SRV and a new time period (i.e. the segments drawn with "=") it uses the current time period and the previous SRV for its first descriptor: that's meant to cover clients with an up-to-date consensus in the same time period as the service.

Example:

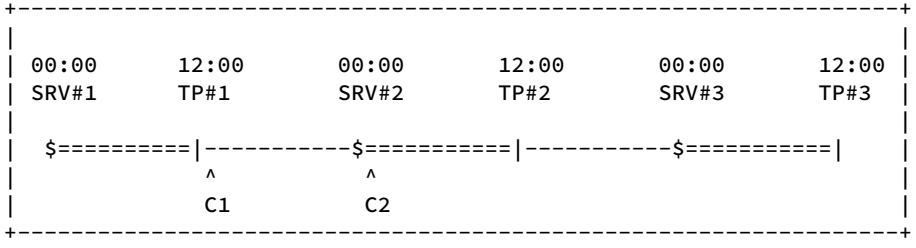


Client behavior for fetching descriptors

And here is how clients use TPs and SRVs to fetch descriptors:

Clients always aim to synchronize their TP with SRV, so they always want to use TP#N with SRV#N: To achieve this wrt time periods, clients always use the current time period when fetching descriptors. Now wrt SRVs, if a client is in the time segment between a new time period and a new SRV (i.e. the segments drawn with "-") it uses the current SRV, else if the client is in a time segment between a new SRV and a new time period (i.e. the segments drawn with "="), it uses the previous SRV.

Example:



The default is "ipv4-preferred".

Any unrecognized value SHOULD be treated as "ipv4-preferred".

(Arti only.)

`Tor-Capabilities`: Determining specific extensions

(Response only.)

In responses, proxies SHOULD include a `Tor-Capabilities` header containing a space-separated list of capabilities. Clients MAY use this header to decide whether a required capability is present.

All new features that we add will have a corresponding capability. Clients SHOULD check for these capabilities whenever they are using a feature which, if absent, would be insecure.

Every request header has a capability of the same name, to indicate that the header will be recognized and processed if sent.

> NOTE: As of 18 Nov 2025 the above paragraph is not implemented. See arti#2259 and tor#41165.

(Supported by Arti, and by C Tor since 0.4.9.4-alpha.)

Stream isolation

The isolation value of an HTTP CONNECT stream is a tuple of its "Proxy-Authorization" header (if any), its "Tor-Stream-Isolation" header (if any), and its "X-Tor-Stream-Isolation" header (if any).

> When possible, applications should prefer `Tor-Stream-Isolation`, then `X-Tor-Stream-Isolation` `Proxy-Authorization` is supported in this role only to work with HTTP libraries that make it difficult to set arbitrary headers.

In contrast to the HTTP spec, implementations MAY ignore HTTP space-folding rules when comparing these headers for equality.

Two HTTP CONNECT streams are not allowed to share a circuit if they have different isolation values.

For stream isolation purposes, C Tor considers a “Proxy-Authentication: X”, “X-Tor-Stream-Isolation: Y” tuple to be equivalent to SOCKS5 authentication with username=X and password=Y.

We are unlikely to preserve this equivalence in Arti without further examination.

For more information on stream isolation, including other factors that can prevent streams from sharing circuits, see proposal 171.

TODO: Integrate proposal 171 this into the spec. (See torspec#269.)

---

1. In versions of Tor before 0.4.9.4-alpha, RESOLVEFAILED and NOROUTE produced status code 404. ↩ ↩2

2. A regular DONE reason, if it arrives after a CONNECTED message, won’t trigger a status line, since the CONNECTED message already triggered a “200 OK” status line. ↩

3. Applications should be unable to trigger a NOTADIRECTORY reason, since it should only happen in response to a BEGINDIR message. ↩

the system better tolerate disappearing HSDirs, hsdir_spread_fetch may be less than hsdir_spread_store.) Again, nodes from lower-numbered replicas are disregarded when choosing the spread for a replica.

Using time periods and SRVs to fetch/upload HS descriptors

Hidden services and clients need to make correct use of time periods (TP) and shared random values (SRVs) to successfully fetch and upload descriptors. Furthermore, to avoid problems with skewed clocks, both clients and services use the ‘valid-after’ time of a live consensus as a way to take decisions with regards to uploading and fetching descriptors. By using the consensus times as the ground truth here, we minimize the desynchronization of clients and services due to system clock. Whenever time-based decisions are taken in this section, assume that they are consensus times and not system times.

As [PUB-SHAREDRANDOM] specifies, consensuses contain two shared random values (the current one and the previous one). Hidden services and clients are asked to match these shared random values with descriptor time periods and use the right SRV when fetching/uploading descriptors. This section attempts to precisely specify how this works.

Let’s start with an illustration of the system:

```
        hsdir_n_replicas = an integer in range [1,16] with default
value 2.
        hsdir_spread_fetch = an integer in range [1,128] with default
value 3.
        hsdir_spread_store = an integer in range [1,128] with default
value 4.
           (Until 0.3.2.8-rc, the default was 3.)
```

To determine where a given hidden service descriptor will be stored in a given period, after the blinded public key for that period is derived, the uploading or downloading party calculates:

```
for replicanum in 1...hsdir_n_replicas:
    hs_service_index(replicanum) =
                    SHA3_256("store-at-idx" |
                             blinded_public_key |
                             INT_8(replicanum) |
                             INT_8(period_length) |
                             INT_8(period_num) )
```

where blinded_public_key is specified in section [KEYBLIND], period_length is the length of the time period in minutes, and period_num is calculated using the current consensus "valid-after" as specified in section [TIME-PERIODS].

Then, for each node listed in the current consensus with the HSDir flag, we compute a directory index for that node as:

```
   hs_relay_index(node) =
                SHA3_256("node-idx" | node_identity |
                         shared_random_value |
                         INT_8(period_num) |
                         INT_8(period_length) )
```

where shared_random_value is the shared value generated by the authorities in section [PUB-SHAREDRANDOM], and node_identity is the ed25519 identity key of the node.

Finally, for replicanum in 1...hsdir_n_replicas, the hidden service host uploads descriptors to the first hsdir_spread_store nodes whose indices immediately follow hs_service_index(replicanum). If any of those nodes have already been selected for a lower-numbered replica of the service, any nodes already chosen are disregarded (i.e. skipped over) when choosing a replica's hsdir_spread_store nodes.

When choosing an HSDir to download from, clients choose randomly from among the first hsdir_spread_fetch nodes after the indices. (Note that, in order to make

Special Hostnames in Tor

Overview

Most of the time, Tor treats user-specified hostnames as opaque: When the user connects to <www.torproject.org>, Tor picks an exit node and uses that node to connect to "www.torproject.org". Some hostnames, however, can be used to override Tor's default behavior and circuit-building rules.

These hostnames can be passed to Tor as the address part of a SOCKS4a or SOCKS5 request. If the application is connected to Tor using an IP-only method (such as SOCKS4, TransPort, or NATDPort), these hostnames can be substituted for certain IP addresses using the MapAddress configuration option or the MAPADDRESS control command.

.exit

```
SYNTAX:  [hostname].[name-or-digest].exit
         [name-or-digest].exit
```

Hostname is a valid hostname; [name-or-digest] is either the nickname of a Tor node or the hex-encoded SHA-1 digest of that node's RSA identity key (`SHA1(DER(KP_relayid_rsa))`).

When Tor sees an address in this format, it uses the specified hostname as the exit node. If no "hostname" component is given, Tor defaults to the published IPv4 address of the exit node.

It is valid to try to resolve hostnames, and in fact upon success Tor will cache an internal mapaddress of the form "www.google.com.foo.exit=64.233.161.99.foo.exit" to speed subsequent lookups.

The .exit notation is disabled by default as of Tor 0.2.2.1-alpha, due to potential application-level attacks.

```
   EXAMPLES:
      www.example.com.exampletornode.exit

         Connect to www.example.com from the node called
 "exampletornode".

      exampletornode.exit

         Connect to the published IP address of "exampletornode" using
         "exampletornode" as the exit.
```

.onion

```
   SYNTAX:  [onion_address].onion
            [ignored].[onion_address].onion
```

For version 3 onion service addresses, `onion_address` is defined as:

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

      where:
        - PUBKEY is the 32-byte ed25519 master pubkey (KP_hs_id)
          of the onion 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
```

When Tor sees an address in this format, it tries to look up and connect to the specified onion service. See rend-spec-v3.txt for full details.

The “ignored” portion of the address is intended for use in vhosting.

When to publish a hidden service descriptor

Hidden services periodically publish their descriptor to the responsible HSDirs. The set of responsible HSDirs is determined as specified in [WHERE-HSDESC].

Specifically, every time a hidden service publishes its descriptor, it also sets up a timer for a random time between 60 minutes and 120 minutes in the future. When the timer triggers, the hidden service needs to publish its descriptor again to the responsible HSDirs for that time period. [TODO: Control republish period using a consensus parameter?]

Overlapping descriptors

Hidden services need to upload multiple descriptors so that they can be reachable to clients with older or newer consensuses than them. Services need to upload their descriptors to the HSDirs *before* the beginning of each upcoming time period, so that they are readily available for clients to fetch them. Furthermore, services should keep uploading their old descriptor even after the end of a time period, so that they can be reachable by clients that still have consensuses from the previous time period.

Hence, services maintain two active descriptors at every point. Clients on the other hand, don’t have a notion of overlapping descriptors, and instead always download the descriptor for the current time period and shared random value. It’s the job of the service to ensure that descriptors will be available for all clients. See section [FETCHUPLOADDESC] for how this is achieved.

[TODO: What to do when we run multiple hidden services in a single host?]

Where to publish a hidden service descriptor

This section specifies how the HSDir hash ring is formed at any given time. Whenever a time value is needed (e.g. to get the current time period number), we assume that clients and services use the valid-after time from their latest live consensus.

The following consensus parameters control where a hidden service descriptor is stored;

```
          * the current time period,
          * the daily subcredential,
          * the hidden service directories' public keys,
          * a shared random value that changes in each time period,
            shared_random_value.
          * a set of network-wide networkstatus consensus parameters.
            (Consensus parameters are integer values voted on by
authorities
            and published in the consensus documents, described in
            dir-spec.txt, section 3.3.)

   Below we explain in more detail.
```

Dividing time into periods

To prevent a single set of hidden service directory from becoming a target by adversaries looking to permanently censor a hidden service, hidden service descriptors are uploaded to different locations that change over time.

The length of a “time period” is controlled by the consensus parameter ‘hsdir-interval’, and is a number of minutes between 30 and 14400 (10 days). The default time period length is 1440 (one day).

Time periods start at the Unix epoch (Jan 1, 1970), and are computed by taking the number of minutes since the epoch and dividing by the time period. However, we want our time periods to start at a regular offset from the SRV voting schedule, so we subtract a “rotation time offset” of 12 voting periods from the number of minutes since the epoch, before dividing by the time period (effectively making “our” epoch start at Jan 1, 1970 12:00UTC when the voting period is 1 hour.)

Example: If the current time is 2016-04-13 11:15:01 UTC, making the seconds since the epoch 1460546101, and the number of minutes since the epoch 24342435. We then subtract the “rotation time offset” of 12*60 minutes from the minutes since the epoch, to get 24341715. If the current time period length is 1440 minutes, by doing the division we see that we are currently in time period number 16903.

Specifically, time period #16903 began 16903*1440*60 + (12*60*60) seconds after the epoch, at 2016-04-12 12:00 UTC, and ended at 16904*1440*60 + (12*60*60) seconds after the epoch, at 2016-04-13 12:00 UTC.

Tor Rendezvous Specification - Version 3

This document specifies how the hidden service version 3 protocol works. This text used to be proposal 224-rend-spec-ng.txt.

This document describes a proposed design and specification for hidden services in Tor version 0.2.5.x or later. It’s a replacement for the current rend-spec.txt, rewritten for clarity and for improved design.

Hidden services: overview and preliminaries

Hidden services aim to provide responder anonymity for bidirectional stream-based communication on the Tor network. Unlike regular Tor connections, where the connection initiator receives anonymity but the responder does not, hidden services attempt to provide bidirectional anonymity.

Participants:

Operator – A person running a hidden service

```
      Host, "Server" -- The Tor software run by the operator to
provide
         a hidden service.

      User -- A person contacting a hidden service.

      Client -- The Tor software running on the User's computer

      Hidden Service Directory (HSDir) -- A Tor node that hosts signed
        statements from hidden service hosts so that users can make
        contact with them.

      Introduction Point -- A Tor node that accepts connection
requests
        for hidden services and anonymously relays those requests to
the
        hidden service.

      Rendezvous Point -- A Tor node to which clients and servers
        connect and which relays traffic between them.
```

Improvements over previous versions

Here is a list of improvements of this proposal over the legacy hidden services:

a) Better crypto (replaced SHA1/DH/RSA1024 with SHA3/ed25519/curve25519) b) Improved directory protocol leaking less to directory servers. c) Improved directory protocol with smaller surface for targeted attacks. d) Better onion address security against impersonation. e) More extensible introduction/ rendezvous protocol. f) Offline keys for onion services g) Restricted discovery mode

Deriving blinded keys and subcredentials

In each time period (see [TIME-PERIODS] for a definition of time periods), a hidden service host uses a different blinded private key to sign its directory information, and clients use a different blinded public key as the index for fetching that information.

For a candidate for a key derivation method, see Appendix [KEYBLIND].

Additionally, clients and hosts derive a subcredential for each period. Knowledge of the subcredential is needed to decrypt hidden service descriptors for each period and to authenticate with the hidden service host in the introduction process. Unlike the credential, it changes each period. Knowing the subcredential does not enable the hidden service host to derive the main credential–therefore, it is safe to put the subcredential on the hidden service host while leaving the hidden service's private key offline.

The subcredential for a period is derived as:

```
N_hs_subcred = SHA3_256("subcredential" | N_hs_cred | blinded-public-
key).
```

In the above formula, credential corresponds to:

```
N_hs_cred = SHA3_256("credential" | public-identity-key)
```

where `public-identity-key` is the public identity master key of the hidden service.

Locating, uploading, and downloading hidden service descriptors

To avoid attacks where a hidden service's descriptor is easily targeted for censorship, we store them at different directories over time, and use shared random values to prevent those directories from being predictable far in advance.

Which Tor servers hosts a hidden service depends on:

Generating and publishing hidden service descriptors [HSDIR]

Hidden service descriptors follow the same metaformat as other Tor directory objects. They are published anonymously to Tor servers with the HSDir flag, the HSDir=2 subprotocol, and tor version >= 0.3.0.8 (because a bug was fixed in this version).

Note: “restricted discovery mode” was previously called “client authorization”

Notation and vocabulary

Unless specified otherwise, all multi-octet integers are big-endian.

We write sequences of bytes in two ways:

1. A sequence of two-digit hexadecimal values in square brackets, as in [AB AD 1D EA].

2. A string of characters enclosed in quotes, as in "Hello". The characters in these strings are encoded in their ascii representations; strings are NOT nul-terminated unless explicitly described as NUL terminated.

We use the words "byte" and "octet" interchangeably.

We use the vertical bar | to denote concatenation.

We use INT_N(val) to denote the network (big-endian) encoding of the unsigned integer “val” in N bytes. For example, INT_4(1337) is [00 00 05 39]. Values are truncated like so: val % (2 ^ (N * 8)). For example, INT_4(42) is 42 % 4294967296 (32 bit).

Cryptographic building blocks

This specification uses the following cryptographic building blocks:

```
      * A stream cipher STREAM(iv, k) where iv is a nonce of length
        S_IV_LEN bytes and k is a key of length S_KEY_LEN bytes.

      * A public key signature system SIGN_KEYGEN()->seckey, pubkey;
        SIGN_SIGN(seckey,msg)->sig; and SIGN_CHECK(pubkey, sig, msg)
->
        { "OK", "BAD" }; where secret keys are of length
SIGN_SECKEY_LEN
        bytes, public keys are of length SIGN_PUBKEY_LEN bytes, and
        signatures are of length SIGN_SIG_LEN bytes.

        This signature system must also support key blinding
operations
        as discussed in appendix [KEYBLIND] and in section [SUBCRED]:
        SIGN_BLIND_SECKEY(seckey, blind)->seckey2 and
        SIGN_BLIND_PUBKEY(pubkey, blind)->pubkey2 .

      * A public key agreement system "PK", providing
        PK_KEYGEN()->seckey, pubkey; PK_VALID(pubkey) -> {"OK",
"BAD"};
        and PK_HANDSHAKE(seckey, pubkey)->output; where secret keys
are
        of length PK_SECKEY_LEN bytes, public keys are of length
        PK_PUBKEY_LEN bytes, and the handshake produces outputs of
        length PK_OUTPUT_LEN bytes.

      * A cryptographic hash function H(d), which must be preimage and
        collision resistant.

      * A cryptographic message authentication code MAC(key,msg) that
        produces outputs of length MAC_LEN bytes.

      * A key derivation function KDF(message, n) that outputs n
bytes.

   As a first pass, I suggest:

      * Instantiate STREAM with AES256-CTR.

      * Instantiate SIGN with Ed25519 and the blinding protocol in
        [KEYBLIND].

      * Instantiate PK with Curve25519.

      * Instantiate our hash H(d) function with SHA3-256.

      * Instantiate KDF with SHAKE256.
        We define SHAKE256_KDF(message,n) as SHAKE256(message, n*8).
        (The SHAKE256 spec defines SHAKE's output length in bits,
        but in the Tor Specifications we generally state lengths in
bytes.)

      * Instantiate MAC(key=k, message=m) with SHA3_256(k_len | k |
```

and without them the introduction to the hidden service cannot be completed. See INTRO-AUTH. `KP_hsc_intro_auth`, `KS_hsc_intro_auth`.

The right way to exchange these keys is to have the client generate keys and send the corresponding public keys to the hidden service out-of-band. An easier but less secure way of doing this exchange would be to have the hidden service generate the keypairs and pass the corresponding private keys to its clients. See section [RESTRICTED-DISCOVERY-MGMT] for more details on how these keys should be managed.

[TODO: Also specify stealth restricted discovery.]

```
      Ephemeral descriptor encryption key -- A short-lived encryption
        keypair made by the service, and used to encrypt the inner
layer
        of hidden service descriptors when restricted discovery is in
        use.
        KP_hss_desc_enc, KS_hss_desc_enc


   Nonces defined in this document:

      N_hs_desc_enc -- a nonce used to derive keys to decrypt the
inner
        encryption layer of hidden service descriptors.  This is
        sometimes also called a "descriptor cookie".

   Public/private keypairs defined elsewhere:

      Onion key -- Short-term encryption keypair (KS_ntor, KP_ntor).

      (Node) identity key (KP_relayid).

   Symmetric key-like things defined elsewhere:

      KH from circuit handshake -- An unpredictable value derived as
      part of the Tor circuit extension handshake, used to tie a
request
      to a particular circuit.
```

In even more detail: Restricted discovery keys

When restricted discovery is enabled, each authorized client of a hidden service has two more asymmetric keypairs which are shared with the hidden service. An entity without those keys is not able to use the hidden service. Throughout this document, we assume that these pre-shared keys are exchanged between the hidden service and its clients in a secure out-of-band fashion.

Specifically, each authorized client possesses:

- (NOTE: implemented as of 0.3.5.1-alpha.) An x25519 keypair used to compute decryption keys that allow the client to decrypt the hidden service descriptor. See HS-DESC-ENC. This is the client's counterpart to `KP_hss_desc_enc`. `KP_hsc_desc_enc`, `KS_hsd_desc_enc`.
- (NOTE: *not* implemented) An ed25519 keypair which allows the client to compute signatures which prove to the hidden service that the client is authorized. These signatures are inserted into the INTRODUCE1 message,

```
m),
        where k_len is htonll(len(k)).
```

When we need a particular MAC key length below, we choose `MAC_KEY_LEN=32` (256 bits).

For legacy purposes, we specify compatibility with older versions of the Tor introduction point and rendezvous point protocols. These used RSA1024, DH1024, AES128, and SHA1, as discussed in rend-spec.txt.

As in [proposal 220], all signatures are generated not over strings themselves, but over those strings prefixed with a distinguishing value.

Protocol building blocks

In sections below, we need to transmit the locations and identities of Tor nodes. We do so in the link identification format used by EXTEND2 messages in the Tor protocol.

```
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]
```

Link specifier types are as described in tor-spec.txt. Every set of link specifiers SHOULD include at minimum specifiers of type [00] (TLS-over-TCP, IPv4), [02] (legacy node identity) and [03] (ed25519 identity key). Sets of link specifiers without these three types SHOULD be rejected.

As of 0.4.1.1-alpha, Tor includes both IPv4 and IPv6 link specifiers in v3 onion service protocol link specifier lists. All available addresses SHOULD be included as link specifiers, regardless of the address that Tor actually used to connect/extend to the remote relay.

We also incorporate Tor's circuit extension handshakes, as used in the CREATE2 and CREATED2 cells described in tor-spec.txt. In these handshakes, a client who knows a public key for a server sends a message and receives a message from that server. Once the exchange is done, the two parties have a shared set of forward-secure key material, and the client knows that nobody else shares that key material unless they control the secret key corresponding to the server's public key.

Assigned relay message types

These relay message types are reserved for use in the hidden service protocol.

32 – RELAY_COMMAND_ESTABLISH_INTRO

```
      Blinded signing key -- A keypair derived from the identity key,
        used to sign descriptor signing keys. It changes periodically
for
        each service. Clients who know a 'credential' consisting of
the
        service's public identity key and an optional secret can
derive
        the public blinded identity key for a service.  This key is
used
        as an index in the DHT-like structure of the directory system
        (see [SUBCRED]).
        KP_hs_blind_id, KS_hs_blind_id.



      Descriptor signing key -- A key used to sign hidden service
        descriptors.  This is signed by blinded signing keys. Unlike
        blinded signing keys and master identity keys, the secret part
        of this key must be stored online by hidden service hosts. The
        public part of this key is included in the unencrypted section
        of HS descriptors (see [DESC-OUTER]).
        KP_hs_desc_sign, KS_hs_desc_sign.



      Introduction point authentication key -- A short-term signing
        keypair used to identify a hidden service's session at a given
        introduction point. The service makes a fresh keypair for each
        introduction point; these are used to sign the request that a
        hidden service host makes when establishing an introduction
        point, so that clients who know the public component of this
key
        can get their introduction requests sent to the right
        service. No keypair is ever used with more than one
introduction
        point. (previously called a "service key" in rend-spec.txt)
        KP_hs_ipt_sid, KS_hs_ipt_sid
        ("hidden service introduction point session id").



      Introduction point encryption key -- A short-term encryption
        keypair used when establishing connections via an introduction
        point. Plays a role analogous to Tor nodes' onion keys. The
service
        makes a fresh keypair for each introduction point.
        KP_hss_ntor, KS_hss_ntor.
```

see [DESC-OUTER] and [HS-DESC-ENC] below), and their corresponding descriptor encryption keys, and export those to the hidden service hosts.

As a result, in the scenario where the Hidden Service gets compromised, the adversary can only impersonate it for a limited period of time (depending on how many signing keys were generated in advance).

It's important to not send the private part of the blinded signing key to the Hidden Service since an attacker can derive from it the secret master identity key. The secret blinded signing key should only be used to create credentials for the descriptor signing keys.

(NOTE: although the protocol allows them, offline keys are not implemented as of 0.3.2.1-alpha.)

In more detail: Encryption Keys And Replay Resistance

To avoid replays of an introduction request by an introduction point, a hidden service host must never accept the same request twice. Earlier versions of the hidden service design used an authenticated timestamp here, but including a view of the current time can create a problematic fingerprint. (See proposal 222 for more discussion.)

In more detail: A menagerie of keys

[In the text below, an "encryption keypair" is roughly "a keypair you can do Diffie-Hellman with" and a "signing keypair" is roughly "a keypair you can do ECDSA with."]

Public/private keypairs defined in this document:

```
Master (hidden service) identity key -- A master signing keypair
  used as the identity for a hidden service.  This key is long
  term and not used on its own to sign anything; it is only used
  to generate blinded signing keys as described in [KEYBLIND]
  and [SUBCRED]. The public key is encoded in the ".onion"
  address according to [NAMING].
  KP_hs_id, KS_hs_id.
```

```
        Sent from hidden service host to introduction point;
        establishes introduction point. Discussed in
        [REG_INTRO_POINT].

    33 -- RELAY_COMMAND_ESTABLISH_RENDEZVOUS

        Sent from client to rendezvous point; creates rendezvous
        point. Discussed in [EST_REND_POINT].

    34 -- RELAY_COMMAND_INTRODUCE1

        Sent from client to introduction point; requests
        introduction. Discussed in [SEND_INTRO1]

    35 -- RELAY_COMMAND_INTRODUCE2

        Sent from introduction point to hidden service host;
requests
        introduction. Same format as INTRODUCE1. Discussed in
        [FMT_INTRO1] and [PROCESS_INTRO2]

    36 -- RELAY_COMMAND_RENDEZVOUS1

        Sent from hidden service host to rendezvous point;
        attempts to join host's circuit to
        client's circuit. Discussed in [JOIN_REND]

    37 -- RELAY_COMMAND_RENDEZVOUS2

        Sent from rendezvous point to client;
        reports join of host's circuit to
        client's circuit. Discussed in [JOIN_REND]

    38 -- RELAY_COMMAND_INTRO_ESTABLISHED

        Sent from introduction point to hidden service host;
        reports status of attempt to establish introduction
        point. Discussed in [INTRO_ESTABLISHED]

    39 -- RELAY_COMMAND_RENDEZVOUS_ESTABLISHED

        Sent from rendezvous point to client; acknowledges
        receipt of ESTABLISH_RENDEZVOUS message. Discussed in
        [EST_REND_POINT]

    40 -- RELAY_COMMAND_INTRODUCE_ACK

        Sent from introduction point to client; acknowledges
        receipt of INTRODUCE1 message and reports success/failure.
        Discussed in [INTRO_ACK]
```

Acknowledgments

This design includes ideas from many people, including

```
Christopher Baines,
Daniel J. Bernstein,
Matthew Finkel,
Ian Goldberg,
George Kadianakis,
Aniket Kate,
Tanja Lange,
Robert Ransom,
Roger Dingledine,
Aaron Johnson,
Tim Wilson-Brown ("teor"),
special (John Brooks),
s7r
```

It's based on Tor's original hidden service design by Roger Dingledine, Nick Mathewson, and Paul Syverson, and on improvements to that design over the years by people including

```
Tobias Kamm,
Thomas Lauterbach,
Karsten Loesing,
Alessandro Preite Martinez,
Robert Ransom,
Ferdinand Rieger,
Christoph Weingarten,
Christian Wilms,
```

We wouldn't be able to do any of this work without good attack designs from researchers including

```
Alex Biryukov,
Lasse Øverlier,
Ivan Pustogarov,
Paul Syverson,
Ralf-Philipp Weinmann,
```

```
See [ATTACK-REFS] for their papers.

Several of these ideas have come from conversations with

   Christian Grothoff,
   Brian Warner,
   Zooko Wilcox-O'Hearn,
```

And if this document makes any sense at all, it's thanks to editing help from

This is achieved using two nonces:

```
    * A "credential", derived from the public identity key KP_hs_id.
      N_hs_cred.

    * A "subcredential", derived from the credential N_hs_cred
      and information which various with the current time period.
      N_hs_subcred.
```

The body of each descriptor is also encrypted with a key derived from the public signing key.

To avoid a "thundering herd" problem where every service generates and uploads a new descriptor at the start of each period, each descriptor comes online at a time during the period that depends on its blinded signing key. The keys for the last period remain valid until the new keys come online.

In more detail: Scaling to multiple hosts

This design is compatible with our current approaches for scaling hidden services. Specifically, hidden service operators can use onionbalance to achieve high availability between multiple nodes on the HSDir layer. Furthermore, operators can use proposal 255 to load balance their hidden services on the introduction layer. See [SCALING-REFS] for further discussions on this topic and alternative designs.

```
1.6. In more detail: Backward compatibility with older hidden service
     protocols
```

This design is incompatible with the clients, server, and hsdir node protocols from older versions of the hidden service protocol as described in rend-spec.txt. On the other hand, it is designed to enable the use of older Tor nodes as rendezvous points and introduction points.

In more detail: Keeping crypto keys offline

In this design, a hidden service's secret identity key may be stored offline. It's used only to generate blinded signing keys, which are used to sign descriptor signing keys.

In order to operate a hidden service, the operator can generate in advance a number of blinded signing keys and descriptor signing keys (and their credentials;

To learn the introduction points, clients must decrypt the body of the hidden service descriptor. To do so, clients must know the *unblinded* public key of the service, which makes the descriptor unusable by entities without that knowledge (e.g. HSDirs that don't know the onion address).

Also, if optional restricted discovery mode is enabled, hidden service descriptors are superencrypted using each authorized user's identity x25519 key, to further ensure that unauthorized entities cannot decrypt it.

In order to make the introduction point send a rendezvous request to the service, the client needs to use the per-introduction-point authentication key found in the hidden service descriptor.

The final level of access control happens at the server itself, which may decide to respond or not respond to the client's request depending on the contents of the request. The protocol is extensible at this point: at a minimum, the server requires that the client demonstrate knowledge of the contents of the encrypted portion of the hidden service descriptor. If optional restricted discovery mode is enabled, the service may additionally require the client to prove knowledge of a pre-shared private key.

In more detail: Distributing hidden service descriptors.

Periodically, hidden service descriptors become stored at different locations to prevent a single directory or small set of directories from becoming a good DoS target for removing a hidden service.

For each period, the Tor directory authorities agree upon a collaboratively generated random value. (See section 2.3 for a description of how to incorporate this value into the voting practice; generating the value is described in other proposals, including [SHAREDRANDOM-REFS].) That value, combined with hidden service directories' public identity keys, determines each HSDir's position in the hash ring for descriptors made in that period.

Each hidden service's descriptors are placed into the ring in positions based on the key that was used to sign them. Note that hidden service descriptors are not signed with the services' public keys directly. Instead, we use a key-blinding system [KEYBLIND] to create a new key-of-the-day for each hidden service. Any client that knows the hidden service's public identity key can derive these blinded signing keys for a given period. It should be impossible to derive the blinded signing key lacking that knowledge.

```
Matthew Finkel,
George Kadianakis,
Peter Palfrader,
Tim Wilson-Brown ("teor"),
```

[XXX Acknowledge the huge bunch of people working on 8106.] [XXX Acknowledge the huge bunch of people working on 8244.]

Please forgive me if I've missed you; please forgive me if I've misunderstood your best ideas here too.

Protocol overview

In this section, we outline the hidden service protocol. This section omits some details in the name of simplicity; those are given more fully below, when we specify the protocol in more detail.

View from 10,000 feet

A hidden service host prepares to offer a hidden service by choosing several Tor nodes to serve as its introduction points. It builds circuits to those nodes, and tells them to forward introduction requests to it using those circuits.

Once introduction points have been picked, the host builds a set of documents called “hidden service descriptors” (or just “descriptors” for short) and uploads them to a set of HSDir nodes. These documents list the hidden service’s current introduction points and describe how to make contact with the hidden service.

When a client wants to connect to a hidden service, it first chooses a Tor node at random to be its “rendezvous point” and builds a circuit to that rendezvous point. If the client does not have an up-to-date descriptor for the service, it contacts an appropriate HSDir and requests such a descriptor.

The client then builds an anonymous circuit to one of the hidden service’s introduction points listed in its descriptor, and gives the introduction point an introduction request to pass to the hidden service. This introduction request includes the target rendezvous point and the first part of a cryptographic handshake.

Upon receiving the introduction request, the hidden service host makes an anonymous circuit to the rendezvous point and completes the cryptographic handshake. The rendezvous point connects the two circuits, and the cryptographic handshake gives the two parties a shared key and proves to the client that it is indeed talking to the hidden service.

Once the two circuits are joined, the client can use Tor relay cells to deliver relay messages to the server: Whenever the rendezvous point receives as relay cell from one of the circuits, it transmits it to the other. (It accepts both RELAY and RELAY_EARLY cells, and retransmits them all as RELAY cells.)

The two parties use these relay messages to implement Tor’s usual application stream protocol: RELAY_BEGIN messages open streams to an external process or processes configured by the server; RELAY_DATA messages are used to communicate data on those streams, and so forth.

In more detail: naming hidden services

A hidden service’s name is its long term master identity key. This is encoded as a hostname by encoding the entire key in Base 32, including a version byte and a checksum, and then appending the string “.onion” at the end. The result is a 56-character domain name.

(This is a change from older versions of the hidden service protocol, where we used an 80-bit truncated SHA1 hash of a 1024 bit RSA key.)

The names in this format are distinct from earlier names because of their length. An older name might look like:

```
      unlikelynamefora.onion
      yyhws9optuwiwsns.onion

   And a new name following this specification might look like:

      l5satjgud6gucryazcyvyvhuxhr74u6ygigiuyixe3a6ysis67ororad.onion

   Please see section [ONIONADDRESS] for the encoding specification.
```

In more detail: Access control

Access control for a hidden service is imposed at multiple points through the process above. Furthermore, there is also the option to impose additional restricted discovery access control using pre-shared secrets exchanged out-of-band between the hidden service and its clients.

> Note: “restricted discovery mode” was previously called “client authorization”

The first stage of access control happens when downloading HS descriptors. Specifically, in order to download a descriptor, clients must know which blinded signing key was used to sign it. (See the next section for more info on key blinding.)