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.

Table of contents:

0. Hidden services: overview and preliminaries.

0.1. Improvements over previous versions.

0.2. Notation and vocabulary

0.3. Cryptographic building blocks

0.4. Protocol building blocks [BUILDING-BLOCKS]

0.5. Assigned relay cell types

0.6. Acknowledgments

1. Protocol overview

1.1. View from 10,000 feet

1.2. In more detail: naming hidden services [NAMING]

1.3. In more detail: Access control [IMD:AC]

1.4. In more detail: Distributing hidden service descriptors. [IMD:DIST]

1.5. In more detail: Scaling to multiple hosts

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

1.7. In more detail: Keeping crypto keys offline

1.8. In more detail: Encryption Keys And Replay Resistance

1.9. In more detail: A menagerie of keys

1.9.1. In even more detail: Client authorization [CLIENT-AUTH]

2. Generating and publishing hidden service descriptors [HSDIR]

2.1. Deriving blinded keys and subcredentials [SUBCRED]

2.2. Locating, uploading, and downloading hidden service descriptors

2.2.1. Dividing time into periods [TIME-PERIODS]

2.2.2. When to publish a hidden service descriptor [WHEN-HSDESC]

2.2.3. Where to publish a hidden service descriptor [WHERE-HSDESC]

2.2.4. Using time periods and SRVs to fetch/upload HS descriptors

2.2.5. Expiring hidden service descriptors [EXPIRE-DESC]

2.2.6. URLs for anonymous uploading and downloading

2.3. Publishing shared random values [PUB-SHAREDRANDOM]

2.3.1. Client behavior in the absense of shared random values

2.3.2. Hidden services and changing shared random values

2.4. Hidden service descriptors: outer wrapper [DESC-OUTER]

2.5. Hidden service descriptors: encryption format [HS-DESC-ENC]

2.5.1. First layer of encryption [HS-DESC-FIRST-LAYER]

2.5.1.1. First layer encryption logic

2.5.1.2. First layer plaintext format

2.5.1.3. Client behavior

2.5.1.4. Obfuscating the number of authorized clients

2.5.2. Second layer of encryption [HS-DESC-SECOND-LAYER]

2.5.2.1. Second layer encryption keys

2.5.2.2. Second layer plaintext format

2.5.3. Deriving hidden service descriptor encryption keys [HS-DESC-ENCRYPTION-KEYS]

3. The introduction protocol [INTRO-PROTOCOL]

3.1. Registering an introduction point [REG\_INTRO\_POINT]

3.1.1. Extensible ESTABLISH\_INTRO protocol. [EST\_INTRO]

3.1.1.1. Denial-of-Server Defense Extension. [EST\_INTRO\_DOS\_EXT]

3.1.2. Registering an introduction point on a legacy Tor node [LEGACY\_EST\_INTRO]

3.1.3. Acknowledging establishment of introduction point [INTRO\_ESTABLISHED]

3.2. Sending an INTRODUCE1 cell to the introduction point. [SEND\_INTRO1]

3.2.1. INTRODUCE1 cell format [FMT\_INTRO1]

3.2.2. INTRODUCE\_ACK cell format. [INTRO\_ACK]

3.3. Processing an INTRODUCE2 cell at the hidden service. [PROCESS\_INTRO2]

3.3.1. Introduction handshake encryption requirements [INTRO-HANDSHAKE-REQS]

3.3.2. Example encryption handshake: ntor with extra data [NTOR-WITH-EXTRA-DATA]

3.4. Authentication during the introduction phase. [INTRO-AUTH]

3.4.1. Ed25519-based authentication.

4. The rendezvous protocol

4.1. Establishing a rendezvous point [EST\_REND\_POINT]

4.2. Joining to a rendezvous point [JOIN\_REND]

4.2.1. Key expansion

4.3. Using legacy hosts as rendezvous points

5. Encrypting data between client and host

6. Encoding onion addresses [ONIONADDRESS]

7. Open Questions:

-1. Draft notes

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.

Look for the string "TODO" below: it describes gaps or uncertainties

in the design.

Change history:

2013-11-29: Proposal first numbered. Some TODO and XXX items remain.

2014-01-04: Clarify some unclear sections.

2014-01-21: Fix a typo.

2014-02-20: Move more things to the revised certificate format in the

new updated proposal 220.

2015-05-26: Fix two typos.

0. 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.

0.1. 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) Advanced client authorization

0.2. 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).

0.3. Cryptographic building blocks

This specification uses the following cryptographic building blocks:

\* A pseudorandom number generator backed by a strong entropy source.

The output of the PRNG should always be hashed before being posted on

the network to avoid leaking raw PRNG bytes to the network

(see [PRNG-REFS]).

\* 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 should be preimage and

collision resistant. It produces hashes of length HASH\_LEN

bytes.

\* 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 H with SHA3-256.

\* Instantiate KDF with SHAKE-256.

\* Instantiate MAC(key=k, message=m) with H(k\_len | k | m),

where k\_len is htonll(len(k)).

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.

0.4. Protocol building blocks [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 cells 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 MUST include at minimum specifiers of type [00]

(TLS-over-TCP, IPv4), [02] (legacy node identity) and [03] (ed25519

identity key).

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.

0.5. Assigned relay cell types

These relay cell types are reserved for use in the hidden service

protocol.

32 -- RELAY\_COMMAND\_ESTABLISH\_INTRO

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 cell. Discussed in

[EST\_REND\_POINT]

40 -- RELAY\_COMMAND\_INTRODUCE\_ACK

Sent from introduction point to client; acknowledges

receipt of INTRODUCE1 cell and reports success/failure.

Discussed in [INTRO\_ACK]

0.6. 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

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.

1. 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.

1.1. 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 send Tor RELAY cells

to the server. RELAY\_BEGIN cells open streams to an external process

or processes configured by the server; RELAY\_DATA cells are used to

communicate data on those streams, and so forth.

1.2. In more detail: naming hidden services [NAMING]

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.

1.3. In more detail: Access control [IMD:AC]

Access control for a hidden service is imposed at multiple points through

the process above. Furthermore, there is also the option to impose

additional client authorization access control using pre-shared secrets

exchanged out-of-band between the hidden service and its clients.

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.)

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 unuseable by entities

without that knowledge (e.g. HSDirs that don't know the onion address).

Also, if optional client authorization 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 client authorization is enabled, the service may additionally

require the client to prove knowledge of a pre-shared private key.

1.4. In more detail: Distributing hidden service descriptors. [IMD:DIST]

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 credential can derive these blinded

signing keys for a given period. It should be impossible to derive

the blinded signing key lacking that credential.

The body of each descriptor is also encrypted with a key derived from

the credential.

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.

1.5. 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.

1.7. 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; 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.)

1.8. 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.)

1.9. 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].

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]).

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]).

Introduction point authentication key -- A short-term signing

keypair used to identify a hidden service to a given

introduction point. A fresh keypair is made 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)

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. A fresh

keypair is made for each introduction point.

Symmetric keys defined in this document:

Descriptor encryption keys -- A symmetric encryption key used to

encrypt the body of hidden service descriptors. Derived from the

current period and the hidden service credential.

Public/private keypairs defined elsewhere:

Onion key -- Short-term encryption keypair

(Node) identity key

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.

1.9.1. In even more detail: Client authorization keys [CLIENT-AUTH]

When client authorization is enabled, each authorized client of a hidden

service has two more assymetric 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:

- An x25519 keypair used to compute decryption keys that allow the client to

decrypt the hidden service descriptor. See [HS-DESC-ENC].

- 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 cell, and without them the

introduction to the hidden service cannot be completed. See [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 [CLIENT-AUTH-MGMT] for more details on how these

keys should be managed.

[TODO: Also specify stealth client authorization.]

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

2. 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, HSDir=2 protocol version and tor version >= 0.3.0.8 (because a

bug was fixed in this version).

2.1. Deriving blinded keys and subcredentials [SUBCRED]

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, even

in combination with the blinded private key, 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:

subcredential = H("subcredential" | credential | blinded-public-key).

In the above formula, credential corresponds to:

credential = H("credential" | public-identity-key)

where public-identity-key is the public identity master key of the hidden

service.

2.2. Locating, uploading, and downloading hidden service descriptors

[HASHRING]

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:

\* the current time period,

\* the daily subcredential,

\* the hidden service directories' public keys,

\* a shared random value that changes in each time period,

\* 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.

2.2.1. Dividing time into periods [TIME-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 12:00UTC every day, so

we subtract a "rotation time offset" of 12\*60 minutes 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).

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.

2.2.2. When to publish a hidden service descriptor [WHEN-HSDESC]

Hidden services periodically publish their descriptor to the responsible

HSDirs. The set of responsible HSDirs is determined as specified in

[WHERE-HSDESC].

Specifically, everytime 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?]

2.2.2.1. 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?]

2.2.3. Where to publish a hidden service descriptor [WHERE-HSDESC]

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;

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\_index(replicanum) = H("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 HSDirV3 flag,

we compute a directory index for that node as:

hsdir\_index(node) = H("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\_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 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.

2.2.4. Using time periods and SRVs to fetch/upload HS descriptors [FETCHUPLOADDESC]

Hidden services and clients need to make correct use of time periods (TP)

and shared random values (SRVs) to successfuly 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:

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

| |

| 00:00 12:00 00:00 12:00 00:00 12:00 |

| SRV#1 TP#1 SRV#2 TP#2 SRV#3 TP#3 |

| |

| $==========|-----------$===========|-----------$===========| |

| |

| |

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

Legend: [TP#1 = Time Period #1]

[SRV#1 = Shared Random Value #1]

["$" = descriptor rotation moment]

2.2.4.1. Client behavior for fetching descriptors [CLIENTFETCH]

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:

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

| |

| 00:00 12:00 00:00 12:00 00:00 12:00 |

| SRV#1 TP#1 SRV#2 TP#2 SRV#3 TP#3 |

| |

| $==========|-----------$===========|-----------$===========| |

| ^ ^ |

| C1 C2 |

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

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.

2.2.4.2. Service behavior for uploading descriptors [SERVICEUPLOAD]

As discussed above, services maintain two active descriptors at any time. We

call these the "first" and "second" service descriptors. Services rotate

their descriptor everytime 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:

2.2.4.2.1. First descriptor upload logic [FIRSTDESCUPLOAD]

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:

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

| |

| 00:00 12:00 00:00 12:00 00:00 12:00 |

| SRV#1 TP#1 SRV#2 TP#2 SRV#3 TP#3 |

| |

| $==========|-----------$===========|-----------$===========| |

| ^ |

| S |

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

Consider that the service is at 01:00 right after SRV#2: it will upload its

first descriptor using TP#1 and SRV#1.

2.2.4.2.2. Second descriptor upload logic [SECONDDESCUPLOAD]

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:

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

| |

| 00:00 12:00 00:00 12:00 00:00 12:00 |

| SRV#1 TP#1 SRV#2 TP#2 SRV#3 TP#3 |

| |

| $==========|-----------$===========|-----------$===========| |

| ^ |

| S |

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

Consider that the service is at 01:00 right after SRV#2: it will upload its

second descriptor using TP#2 and SRV#2.

2.2.5. Expiring hidden service descriptors [EXPIRE-DESC]

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).

2.2.6. 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.

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

anything else.

2.2.7. 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 natutally) 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].

2.3. Publishing shared random values [PUB-SHAREDRANDOM]

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

2.3.1. Client behavior in the absense 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 = H("shared-random-disaster" | INT\_8(period\_length) | INT\_8(period\_num))

where period\_length is the length of a time period in minutes, period\_num is

calculated as specified in [TIME-PERIODS] for the wanted shared random value

that could not be found originally.

2.3.2. 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.

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?

2.4. 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.

"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.)

"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).

2.5. Hidden service descriptors: encryption format [HS-DESC-ENC]

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 client authorization is enabled

and protects against entities that do not possess valid client credentials.

2.5.1. First layer of encryption [HS-DESC-FIRST-LAYER]

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.

2.5.1.1. 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.

2.5.1.2. 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 client auth is enabled, the hidden service generates a fresh

descriptor\_cookie key (32 random bytes) and encrypts it using each

authorized client's identity x25519 key. Authorized clients can use the

descriptor cookie to decrypt the second layer of encryption. Our encryption

scheme requires the hidden service to also generate an ephemeral x25519

keypair for each new descriptor.

If client auth is disabled, fake data is placed in each of the fields below

to obfuscate whether client authorization is enabled.

Here are all the supported fields:

"desc-auth-type" SP type NL

[Exactly once]

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 client authorization is disabled, the value here should be "x25519".

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

[Exactly once]

This field contains an ephemeral x25519 public key generated by the

hidden service and encoded in base64. The key is used by the encryption

scheme below.

If client authorization 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

[Any number]

When client authorization 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 client authorization is enabled, each "auth-client" line contains

the descriptor cookie encrypted to each individual client. We assume that

each authorized client possesses a pre-shared x25519 keypair which is

used to decrypt the descriptor cookie.

We now describe the descriptor cookie encryption scheme. Here are the

relevant keys:

client\_x = private x25519 key of authorized client

client\_X = public x25519 key of authorized client

hs\_y = private key of ephemeral x25519 keypair of hidden service

hs\_Y = public key of ephemeral x25519 keypair of hidden service

descriptor\_cookie = descriptor cookie used to encrypt the descriptor

And here is what the hidden service computes:

SECRET\_SEED = x25519(hs\_y, client\_X)

KEYS = KDF(subcredential | 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:

- 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 descriptor\_cookie

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.

2.5.1.3. Client behavior [FIRST-LAYER-CLIENT-BEHAVIOR]

The goal of clients at this stage is to decrypt the "encrypted" field as

described in [HS-DESC-SECOND-LAYER].

If client authorization 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].

2.5.1.4. Hiding client authorization data

Hidden services should avoid leaking whether client authorization is

enabled or how many authorized clients there are.

Hence even when client authorization 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).

2.5.2. Second layer of encryption [HS-DESC-SECOND-LAYER]

The second layer of descriptor encryption is designed to protect descriptor

confidentiality against unauthorized clients. If client authorization 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 client authorization is disabled, then the second layer of HS encryption

does not offer any additional security, but is still used.

2.5.2.1. 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:

SECRET\_DATA = blinded-public-key | descriptor\_cookie

STRING\_CONSTANT = "hsdir-encrypted-data"

If client authorization is disabled the 'descriptor\_cookie' field is left blank.

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

2.5.2.2. 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:

"create2-formats" SP formats NL

[Exactly once]

A space-separated list of integers denoting CREATE2 cell format numbers

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.

"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: 'password' and 'ed25519'.

"single-onion-service"

[None or 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.

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.

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 MAY perform basic validity checks on the link

specifiers in the descriptor. These checks SHOULD NOT leak

detailed information about the client's version, configuration,

or consensus. (See 3.3 for service link specifier handling.)

"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.

"auth-key" NL certificate NL

[Exactly once per introduction point]

The certificate is a proposal 220 certificate wrapped in

"-----BEGIN ED25519 CERT-----", cross-certifying the descriptor

signing key with the introduction point authentication key, which

is included in the mandatory signing-key extension. The certificate

type must be [09].

"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.

"enc-key-cert" NL certificate NL

[Exactly once per introduction point]

Cross-certification of the descriptor signing key by the encryption

key.

For "ntor" keys, certificate is a proposal 220 certificate wrapped

in "-----BEGIN ED25519 CERT-----" armor, cross-certifying the

descriptor signing key with the ed25519 equivalent of a curve25519

public encryption key derived using the process in proposal 228

appendix A. The certificate type must be [0B], and the signing-key

extension is mandatory.

"legacy-key" NL key NL

[None or at most once per introduction point]

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 protocol version value

"HSIntro 3".

"legacy-key-cert" NL certificate NL

[None or at most once per introduction point]

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

descriptor signing key with the RSA public key found in

"legacy-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 prroceed with the introduction protocol as specified in [INTRO-PROTOCOL].

2.5.3. Deriving hidden service descriptor encryption keys [HS-DESC-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:

SALT = 16 bytes from H(random), changes each time we rebuld 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 | subcredential | INT\_8(revision\_counter)

keys = 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 MAC of both above fields [32 bytes]

The final encryption format is ENCRYPTED = STREAM(SECRET\_IV,SECRET\_KEY) XOR Plaintext

2.5.4. Number of introduction points [NUM\_INTRO\_POINT]

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.

3. The introduction protocol [INTRO-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.

3.1. Registering an introduction point [REG\_INTRO\_POINT]

3.1.1. Extensible ESTABLISH\_INTRO protocol. [EST\_INTRO]

When a hidden service is establishing a new introduction point, it

sends an ESTABLISH\_INTRO cell with the following contents:

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 cells; 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.

The EXT\_FIELD\_TYPE, EXT\_FIELD\_LEN, EXT\_FIELD entries are reserved for

future extensions to the introduction protocol. Extensions with

unrecognized EXT\_FIELD\_TYPE values must be ignored.

The HANDSHAKE\_AUTH field contains the MAC of all earlier fields in

the cell using as its key the shared per-circuit material ("KH")

generated during the circuit extension protocol; see tor-spec.txt

section 5.2, "Setting circuit keys". It prevents replays of

ESTABLISH\_INTRO cells.

SIG\_LEN is the length of the signature.

SIG is a signature, using AUTH\_KEY, of all contents of the cell, up

to but not including SIG. These contents are prefixed with the string

"Tor establish-intro cell v1".

Upon receiving an ESTABLISH\_INTRO cell, a Tor node first decodes the

key and the signature, and checks the signature. The node must reject

the ESTABLISH\_INTRO cell 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 cells.

3.1.1.1. Denial-of-Service Defense Extension. [EST\_INTRO\_DOS\_EXT]

This extension can be used to send Denial-of-Service (DoS) parameters to

the introduction point in order for it to apply them for the introduction

circuit.

If used, it needs to be encoded within the N\_EXTENSIONS field of the

ESTABLISH\_INTRO cell defined in the previous section. The content is

defined as follow:

EXT\_FIELD\_TYPE:

[01] -- Denial-of-Service Parameters.

If this flag is set, the extension should be used by the introduction

point to learn what values the denial of service subsystem should be

using.

EXT\_FIELD content format is:

N\_PARAMS [1 byte]

N\_PARAMS times:

PARAM\_TYPE [1 byte]

PARAM\_VALUE [8 byte]

The PARAM\_TYPE possible values are:

[01] -- DOS\_INTRODUCE2\_RATE\_PER\_SEC

The rate per second of INTRODUCE2 cell relayed to the

service.

[02] -- DOS\_INTRODUCE2\_BURST\_PER\_SEC

The burst per second of INTRODUCE2 cell relayed to the

service.

The PARAM\_VALUE size is 8 bytes in order to accomodate 64bit values.

It MUST match the specified limit for the following PARAM\_TYPE:

[01] -- Min: 0, Max: 2147483647

[02] -- Min: 0, Max: 2147483647

A value of 0 means the defense is disabled. If the rate per second is

set to 0 (param 0x01) then the burst value should be ignored. And

vice-versa, if the burst value is 0 (param 0x02), then the rate value

should be ignored. In other words, setting one single parameter to 0

disables the defense.

The burst can NOT be smaller than the rate. If so, the parameters

should be ignored by the introduction point.

Any valid value does have precedence over the network wide consensus

parameter.

Using this extension extends the payload of the ESTABLISH\_INTRO cell by 19

bytes bringing it from 134 bytes to 155 bytes.

This extension can only be used with relays supporting the protocol version

"HSIntro=5".

Introduced in tor-0.4.2.1-alpha.

3.1.2. Registering an introduction point on a legacy Tor node

[LEGACY\_EST\_INTRO]

Tor nodes should also support an older version of the ESTABLISH\_INTRO

cell, first documented in rend-spec.txt. New hidden service hosts

must use this format when establishing introduction points at older

Tor nodes that do not support the format above in [EST\_INTRO].

In this older protocol, an ESTABLISH\_INTRO cell contains:

KEY\_LEN [2 bytes]

KEY [KEY\_LEN bytes]

HANDSHAKE\_AUTH [20 bytes]

SIG [variable, up to end of relay payload]

The KEY\_LEN variable determines the length of the KEY field.

The KEY field is the ASN1-encoded legacy RSA public key that was also

included in the hidden service descriptor.

The HANDSHAKE\_AUTH field contains the SHA1 digest of (KH | "INTRODUCE").

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

earlier fields.

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

authentication keys.

3.1.3. Acknowledging establishment of introduction point [INTRO\_ESTABLISHED]

After setting up an introduction circuit, the introduction point reports its

status back to the hidden service host with an INTRO\_ESTABLISHED cell.

The INTRO\_ESTABLISHED cell has the following contents:

N\_EXTENSIONS [1 byte]

N\_EXTENSIONS times:

EXT\_FIELD\_TYPE [1 byte]

EXT\_FIELD\_LEN [1 byte]

EXT\_FIELD [EXT\_FIELD\_LEN bytes]

Older versions of Tor send back an empty INTRO\_ESTABLISHED cell instead.

Services must accept an empty INTRO\_ESTABLISHED cell from a legacy relay.

3.2. Sending an INTRODUCE1 cell to the introduction point. [SEND\_INTRO1]

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

know the following:

\* An introduction point for a service.

\* The introduction authentication key for that introduction point.

\* The introduction encryption key for that introduction point.

The client sends an INTRODUCE1 cell to the introduction point,

containing an identifier for the service, an identifier for the

encryption key that the client intends to use, and an opaque blob to

be relayed to the hidden service host.

In reply, the introduction point sends an INTRODUCE\_ACK cell back to

the client, either informing it that its request has been delivered,

or that its request will not succeed.

[TODO: specify what tor should do when receiving a malformed cell. Drop it?

Kill circuit? This goes for all possible cells.]

3.2.1. INTRODUCE1 cell format [FMT\_INTRO1]

When a client is connecting to an introduction point, INTRODUCE1 cells

should be of the form:

LEGACY\_KEY\_ID [20 bytes]

AUTH\_KEY\_TYPE [1 byte]

AUTH\_KEY\_LEN [2 bytes]

AUTH\_KEY [AUTH\_KEY\_LEN bytes]

N\_EXTENSIONS [1 byte]

N\_EXTENSIONS times:

EXT\_FIELD\_TYPE [1 byte]

EXT\_FIELD\_LEN [1 byte]

EXT\_FIELD [EXT\_FIELD\_LEN bytes]

ENCRYPTED [Up to end of relay payload]

AUTH\_KEY\_TYPE is defined as in [EST\_INTRO]. Currently, the only value of

AUTH\_KEY\_TYPE for this cell is an Ed25519 public key [02].

The LEGACY\_KEY\_ID field is used to distinguish between legacy and new style

INTRODUCE1 cells. In new style INTRODUCE1 cells, LEGACY\_KEY\_ID is 20 zero

bytes. Upon receiving an INTRODUCE1 cell, the introduction point checks the

LEGACY\_KEY\_ID field. If LEGACY\_KEY\_ID is non-zero, the INTRODUCE1 cell

should be handled as a legacy INTRODUCE1 cell by the intro point.

Upon receiving a INTRODUCE1 cell, the introduction point checks

whether AUTH\_KEY matches the introduction point authentication key for an

active introduction circuit. If so, the introduction point sends an

INTRODUCE2 cell with exactly the same contents to the service, and sends an

INTRODUCE\_ACK response to the client.

3.2.2. INTRODUCE\_ACK cell format. [INTRO\_ACK]

An INTRODUCE\_ACK cell has the following fields:

STATUS [2 bytes]

N\_EXTENSIONS [1 bytes]

N\_EXTENSIONS times:

EXT\_FIELD\_TYPE [1 byte]

EXT\_FIELD\_LEN [1 byte]

EXT\_FIELD [EXT\_FIELD\_LEN bytes]

Recognized status values are:

[00 00] -- Success: cell relayed to hidden service host.

[00 01] -- Failure: service ID not recognized

[00 02] -- Bad message format

[00 03] -- Can't relay cell to service

3.3. Processing an INTRODUCE2 cell at the hidden service. [PROCESS\_INTRO2]

Upon receiving an INTRODUCE2 cell, the hidden service host checks whether

the AUTH\_KEY or LEGACY\_KEY\_ID field matches the keys for this

introduction circuit.

The service host then checks whether it has received a cell with these

contents or rendezvous cookie before. If it has, it silently drops it as a

replay. (It must maintain a replay cache for as long as it accepts cells

with the same encryption key. Note that the encryption format below should

be non-malleable.)

If the cell is not a replay, it decrypts the ENCRYPTED field,

establishes a shared key with the client, and authenticates the whole

contents of the cell as having been unmodified since they left the

client. There may be multiple ways of decrypting the ENCRYPTED field,

depending on the chosen type of the encryption key. Requirements for

an introduction handshake protocol are described in

[INTRO-HANDSHAKE-REQS]. We specify one below in section

[NTOR-WITH-EXTRA-DATA].

The decrypted plaintext must have the form:

RENDEZVOUS\_COOKIE [20 bytes]

N\_EXTENSIONS [1 byte]

N\_EXTENSIONS times:

EXT\_FIELD\_TYPE [1 byte]

EXT\_FIELD\_LEN [1 byte]

EXT\_FIELD [EXT\_FIELD\_LEN bytes]

ONION\_KEY\_TYPE [1 bytes]

ONION\_KEY\_LEN [2 bytes]

ONION\_KEY [ONION\_KEY\_LEN bytes]

NSPEC (Number of link specifiers) [1 byte]

NSPEC times:

LSTYPE (Link specifier type) [1 byte]

LSLEN (Link specifier length) [1 byte]

LSPEC (Link specifier) [LSLEN bytes]

PAD (optional padding) [up to end of plaintext]

Upon processing this plaintext, the hidden service makes sure that

any required authentication is present in the extension fields, and

then extends a rendezvous circuit to the node described in the LSPEC

fields, using the ONION\_KEY to complete the extension. As mentioned

in [BUILDING-BLOCKS], the "TLS-over-TCP, IPv4" and "Legacy node

identity" specifiers must be present.

As of 0.4.1.1-alpha, clients include both IPv4 and IPv6 link specifiers

in INTRODUCE1 cells. All available addresses SHOULD be included in the

cell, regardless of the address that the client actually used to extend

to the rendezvous point.

The hidden service should handle invalid or unrecognised link specifiers

the same way as clients do in section 2.5.2.2. In particular, services

MAY perform basic validity checks on link specifiers, and SHOULD NOT

reject unrecognised link specifiers, to avoid information leaks.

The ONION\_KEY\_TYPE field is:

[01] NTOR: ONION\_KEY is 32 bytes long.

The ONION\_KEY field describes the onion key that must be used when

extending to the rendezvous point. It must be of a type listed as

supported in the hidden service descriptor.

When using a legacy introduction point, the INTRODUCE cells must be padded

to a certain length using the PAD field in the encrypted portion.

Upon receiving a well-formed INTRODUCE2 cell, the hidden service host

will have:

\* The information needed to connect to the client's chosen

rendezvous point.

\* The second half of a handshake to authenticate and establish a

shared key with the hidden service client.

\* A set of shared keys to use for end-to-end encryption.

3.3.1. Introduction handshake encryption requirements [INTRO-HANDSHAKE-REQS]

When decoding the encrypted information in an INTRODUCE2 cell, a

hidden service host must be able to:

\* Decrypt additional information included in the INTRODUCE2 cell,

to include the rendezvous token and the information needed to

extend to the rendezvous point.

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

\* Authenticate that the cell has not been modified since the client

generated it.

Note that the old TAP-derived protocol of the previous hidden service

design achieved the first two requirements, but not the third.

3.3.2. Example encryption handshake: ntor with extra data

[NTOR-WITH-EXTRA-DATA]

[TODO: relocate this]

This is a variant of the ntor handshake (see tor-spec.txt, section

5.1.4; see proposal 216; and see "Anonymity and one-way

authentication in key-exchange protocols" by Goldberg, Stebila, and

Ustaoglu).

It behaves the same as the ntor handshake, except that, in addition

to negotiating forward secure keys, it also provides a means for

encrypting non-forward-secure data to the server (in this case, to

the hidden service host) as part of the handshake.

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

the ntor handshake.

The PROTOID for this variant is "tor-hs-ntor-curve25519-sha3-256-1".

We also use the following tweak values:

t\_hsenc = PROTOID | ":hs\_key\_extract"

t\_hsverify = PROTOID | ":hs\_verify"

t\_hsmac = PROTOID | ":hs\_mac"

m\_hsexpand = PROTOID | ":hs\_key\_expand"

To make an INTRODUCE1 cell, the client must know a public encryption

key B for the hidden service on this introduction circuit. The client

generates a single-use keypair:

x,X = KEYGEN()

and computes:

intro\_secret\_hs\_input = EXP(B,x) | AUTH\_KEY | X | B | PROTOID

info = m\_hsexpand | subcredential

hs\_keys = KDF(intro\_secret\_hs\_input | t\_hsenc | info, S\_KEY\_LEN+MAC\_LEN)

ENC\_KEY = hs\_keys[0:S\_KEY\_LEN]

MAC\_KEY = hs\_keys[S\_KEY\_LEN:S\_KEY\_LEN+MAC\_KEY\_LEN]

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

CLIENT\_PK [PK\_PUBKEY\_LEN bytes]

ENCRYPTED\_DATA [Padded to length of plaintext]

MAC [MAC\_LEN bytes]

Substituting those fields into the INTRODUCE1 cell body format

described in [FMT\_INTRO1] above, we have

LEGACY\_KEY\_ID [20 bytes]

AUTH\_KEY\_TYPE [1 byte]

AUTH\_KEY\_LEN [2 bytes]

AUTH\_KEY [AUTH\_KEY\_LEN bytes]

N\_EXTENSIONS [1 bytes]

N\_EXTENSIONS times:

EXT\_FIELD\_TYPE [1 byte]

EXT\_FIELD\_LEN [1 byte]

EXT\_FIELD [EXT\_FIELD\_LEN bytes]

ENCRYPTED:

CLIENT\_PK [PK\_PUBKEY\_LEN bytes]

ENCRYPTED\_DATA [Padded to length of plaintext]

MAC [MAC\_LEN bytes]

(This format is as documented in [FMT\_INTRO1] above, except that here

we describe how to build the ENCRYPTED portion.)

Here, the encryption key plays the role of B in the regular ntor

handshake, and the AUTH\_KEY field plays the role of the node ID.

The CLIENT\_PK field is the public key X. The ENCRYPTED\_DATA field is

the message plaintext, encrypted with the symmetric key ENC\_KEY. The

MAC field is a MAC of all of the cell from the AUTH\_KEY through the

end of ENCRYPTED\_DATA, using the MAC\_KEY value as its key.

To process this format, the hidden service checks PK\_VALID(CLIENT\_PK)

as necessary, and then computes ENC\_KEY and MAC\_KEY as the client did

above, except using EXP(CLIENT\_PK,b) in the calculation of

intro\_secret\_hs\_input. The service host then checks whether the MAC is

correct. If it is invalid, it drops the cell. Otherwise, it computes

the plaintext by decrypting ENCRYPTED\_DATA.

The hidden service host now completes the service side of the

extended ntor handshake, as described in tor-spec.txt section 5.1.4,

with the modified PROTOID as given above. To be explicit, the hidden

service host generates a keypair of y,Y = KEYGEN(), and uses its

introduction point encryption key 'b' to computes:

intro\_secret\_hs\_input = EXP(X,b) | AUTH\_KEY | X | B | PROTOID

info = m\_hsexpand | subcredential

hs\_keys = KDF(intro\_secret\_hs\_input | t\_hsenc | info, S\_KEY\_LEN+MAC\_LEN)

HS\_DEC\_KEY = hs\_keys[0:S\_KEY\_LEN]

HS\_MAC\_KEY = hs\_keys[S\_KEY\_LEN:S\_KEY\_LEN+MAC\_KEY\_LEN]

(The above are used to check the MAC and then decrypt the

encrypted data.)

rend\_secret\_hs\_input = EXP(X,y) | EXP(X,b) | AUTH\_KEY | B | X | Y | PROTOID

NTOR\_KEY\_SEED = MAC(rend\_secret\_hs\_input, t\_hsenc)

verify = MAC(rend\_secret\_hs\_input, t\_hsverify)

auth\_input = verify | AUTH\_KEY | B | Y | X | PROTOID | "Server"

AUTH\_INPUT\_MAC = MAC(auth\_input, t\_hsmac)

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

The server's handshake reply is:

SERVER\_PK Y [PK\_PUBKEY\_LEN bytes]

AUTH AUTH\_INPUT\_MAC [MAC\_LEN bytes]

These fields will be sent to the client in a RENDEZVOUS1 cell using the

HANDSHAKE\_INFO element (see [JOIN\_REND]).

The hidden service host now also knows the keys generated by the

handshake, which it will use to encrypt and authenticate data

end-to-end between the client and the server. These keys are as

computed in tor-spec.txt section 5.1.4.

3.4. Authentication during the introduction phase. [INTRO-AUTH]

Hidden services may restrict access only to authorized users.

One mechanism to do so is the credential mechanism, where only users who

know the credential for a hidden service may connect at all.

3.4.1. Ed25519-based authentication.

To authenticate with an Ed25519 private key, the user must include an

extension field in the encrypted part of the INTRODUCE1 cell with an

EXT\_FIELD\_TYPE type of [02] and the contents:

Nonce [16 bytes]

Pubkey [32 bytes]

Signature [64 bytes]

Nonce is a random value. Pubkey is the public key that will be used

to authenticate. [TODO: should this be an identifier for the public

key instead?] Signature is the signature, using Ed25519, of:

"hidserv-userauth-ed25519"

Nonce (same as above)

Pubkey (same as above)

AUTH\_KEY (As in the INTRODUCE1 cell)

The hidden service host checks this by seeing whether it recognizes

and would accept a signature from the provided public key. If it

would, then it checks whether the signature is correct. If it is,

then the correct user has authenticated.

Replay prevention on the whole cell is sufficient to prevent replays

on the authentication.

Users SHOULD NOT use the same public key with multiple hidden

services.

4. The rendezvous protocol

Before connecting to a hidden service, the client first builds a

circuit to an arbitrarily chosen Tor node (known as the rendezvous

point), and sends an ESTABLISH\_RENDEZVOUS cell. The hidden service

later connects to the same node and sends a RENDEZVOUS cell. Once

this has occurred, the relay forwards the contents of the RENDEZVOUS

cell to the client, and joins the two circuits together.

Single Onion Services attempt to build a non-anonymous single-hop circuit,

but use an anonymous 3-hop circuit if:

\* the rend point is on an address that is configured as unreachable via

a direct connection, or

\* the initial attempt to connect to the rend point over a single-hop

circuit fails, and they are retrying the rend point connection.

4.1. Establishing a rendezvous point [EST\_REND\_POINT]

The client sends the rendezvous point a RELAY\_COMMAND\_ESTABLISH\_RENDEZVOUS

cell containing a 20-byte value.

RENDEZVOUS\_COOKIE [20 bytes]

Rendezvous points MUST ignore any extra bytes in an

ESTABLISH\_RENDEZVOUS cell. (Older versions of Tor did not.)

The rendezvous cookie is an arbitrary 20-byte value, chosen randomly

by the client. The client SHOULD choose a new rendezvous cookie for

each new connection attempt. If the rendezvous cookie is already in

use on an existing circuit, the rendezvous point should reject it and

destroy the circuit.

Upon receiving an ESTABLISH\_RENDEZVOUS cell, the rendezvous point associates

the cookie with the circuit on which it was sent. It replies to the client

with an empty RENDEZVOUS\_ESTABLISHED cell to indicate success. Clients MUST

ignore any extra bytes in a RENDEZVOUS\_ESTABLISHED cell.

The client MUST NOT use the circuit which sent the cell for any

purpose other than rendezvous with the given location-hidden service.

The client should establish a rendezvous point BEFORE trying to

connect to a hidden service.

4.2. Joining to a rendezvous point [JOIN\_REND]

To complete a rendezvous, the hidden service host builds a circuit to

the rendezvous point and sends a RENDEZVOUS1 cell containing:

RENDEZVOUS\_COOKIE [20 bytes]

HANDSHAKE\_INFO [variable; depends on handshake type

used.]

where RENDEZVOUS\_COOKIE is the cookie suggested by the client during the

introduction (see [PROCESS\_INTRO2]) and HANDSHAKE\_INFO is defined in

[NTOR-WITH-EXTRA-DATA].

If the cookie matches the rendezvous cookie set on any

not-yet-connected circuit on the rendezvous point, the rendezvous

point connects the two circuits, and sends a RENDEZVOUS2 cell to the

client containing the HANDSHAKE\_INFO field of the RENDEZVOUS1 cell.

Upon receiving the RENDEZVOUS2 cell, the client verifies that HANDSHAKE\_INFO

correctly completes a handshake. To do so, the client parses SERVER\_PK from

HANDSHAKE\_INFO and reverses the final operations of section

[NTOR-WITH-EXTRA-DATA] as shown here:

rend\_secret\_hs\_input = EXP(Y,x) | EXP(B,x) | AUTH\_KEY | B | X | Y | PROTOID

NTOR\_KEY\_SEED = MAC(ntor\_secret\_input, t\_hsenc)

verify = MAC(ntor\_secret\_input, t\_hsverify)

auth\_input = verify | AUTH\_KEY | B | Y | X | PROTOID | "Server"

AUTH\_INPUT\_MAC = MAC(auth\_input, t\_hsmac)

Finally the client verifies that the received AUTH field of HANDSHAKE\_INFO

is equal to the computed AUTH\_INPUT\_MAC.

Now both parties use the handshake output to derive shared keys for use on

the circuit as specified in the section below:

4.2.1. Key expansion

The hidden service and its client need to derive crypto keys from the

NTOR\_KEY\_SEED part of the handshake output. To do so, they use the KDF

construction as follows:

K = KDF(NTOR\_KEY\_SEED | m\_hsexpand, HASH\_LEN \* 2 + S\_KEY\_LEN \* 2)

The first HASH\_LEN bytes of K form the forward digest Df; the next HASH\_LEN

bytes form the backward digest Db; the next S\_KEY\_LEN bytes form Kf, and the

final S\_KEY\_LEN bytes form Kb. Excess bytes from K are discarded.

Subsequently, the rendezvous point passes relay cells, unchanged, from each

of the two circuits to the other. When Alice's OP sends RELAY cells along

the circuit, it authenticates with Df, and encrypts them with the Kf, then

with all of the keys for the ORs in Alice's side of the circuit; and when

Alice's OP receives RELAY cells from the circuit, it decrypts them with the

keys for the ORs in Alice's side of the circuit, then decrypts them with Kb,

and checks integrity with Db. Bob's OP does the same, with Kf and Kb

interchanged.

[TODO: Should we encrypt HANDSHAKE\_INFO as we did INTRODUCE2

contents? It's not necessary, but it could be wise. Similarly, we

should make it extensible.]

4.3. Using legacy hosts as rendezvous points

The behavior of ESTABLISH\_RENDEZVOUS is unchanged from older versions

of this protocol, except that relays should now ignore unexpected

bytes at the end.

Old versions of Tor required that RENDEZVOUS cell payloads be exactly

168 bytes long. All shorter rendezvous payloads should be padded to

this length with random bytes, to make them difficult to distinguish from

older protocols at the rendezvous point.

Relays older than 0.2.9.1 should not be used for rendezvous points by next

generation onion services because they enforce too-strict length checks to

rendezvous cells. Hence the "HSRend" protocol from proposal#264 should be

used to select relays for rendezvous points.

5. Encrypting data between client and host

A successfully completed handshake, as embedded in the

INTRODUCE/RENDEZVOUS cells, gives the client and hidden service host

a shared set of keys Kf, Kb, Df, Db, which they use for sending

end-to-end traffic encryption and authentication as in the regular

Tor relay encryption protocol, applying encryption with these keys

before other encryption, and decrypting with these keys before other

decryption. The client encrypts with Kf and decrypts with Kb; the

service host does the opposite.

6. Encoding onion addresses [ONIONADDRESS]

The onion address of a hidden service includes its identity public key, a

version field and a basic checksum. All this information is then base32

encoded as shown below:

onion\_address = base32(PUBKEY | CHECKSUM | VERSION) + ".onion"

CHECKSUM = H(".onion checksum" | PUBKEY | VERSION)[:2]

where:

- PUBKEY is the 32 bytes ed25519 master pubkey of the hidden service.

- VERSION is an one byte version field (default value '\x03')

- ".onion checksum" is a constant string

- CHECKSUM is truncated to two bytes before inserting it in onion\_address

Here are a few example addresses:

pg6mmjiyjmcrsslvykfwnntlaru7p5svn6y2ymmju6nubxndf4pscryd.onion

sp3k262uwy4r2k3ycr5awluarykdpag6a7y33jxop4cs2lu5uz5sseqd.onion

xa4r2iadxm55fbnqgwwi5mymqdcofiu3w6rpbtqn7b2dyn7mgwj64jyd.onion

For more information about this encoding, please see our discussion thread

at [ONIONADDRESS-REFS].

7. Open Questions:

Scaling hidden services is hard. There are on-going discussions that

you might be able to help with. See [SCALING-REFS].

How can we improve the HSDir unpredictability design proposed in

[SHAREDRANDOM]? See [SHAREDRANDOM-REFS] for discussion.

How can hidden service addresses become memorable while retaining

their self-authenticating and decentralized nature? See

[HUMANE-HSADDRESSES-REFS] for some proposals; many more are possible.

Hidden Services are pretty slow. Both because of the lengthy setup

procedure and because the final circuit has 6 hops. How can we make

the Hidden Service protocol faster? See [PERFORMANCE-REFS] for some

suggestions.

References:

[KEYBLIND-REFS]:

https://trac.torproject.org/projects/tor/ticket/8106

https://lists.torproject.org/pipermail/tor-dev/2012-September/004026.html

[KEYBLIND-PROOF]:

https://lists.torproject.org/pipermail/tor-dev/2013-December/005943.html

[SHAREDRANDOM-REFS]:

https://gitweb.torproject.org/torspec.git/tree/proposals/250-commit-reveal-consensus.txt

https://trac.torproject.org/projects/tor/ticket/8244

[SCALING-REFS]:

https://lists.torproject.org/pipermail/tor-dev/2013-October/005556.html

[HUMANE-HSADDRESSES-REFS]:

https://gitweb.torproject.org/torspec.git/blob/HEAD:/proposals/ideas/xxx-onion-nyms.txt

http://archives.seul.org/or/dev/Dec-2011/msg00034.html

[PERFORMANCE-REFS]:

"Improving Efficiency and Simplicity of Tor circuit

establishment and hidden services" by Overlier, L., and

P. Syverson

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

[ATTACK-REFS]:

"Trawling for Tor Hidden Services: Detection, Measurement,

Deanonymization" by Alex Biryukov, Ivan Pustogarov,

Ralf-Philipp Weinmann

"Locating Hidden Servers" by Lasse Øverlier and Paul

Syverson

[ED25519-REFS]:

"High-speed high-security signatures" by Daniel

J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and

Bo-Yin Yang. http://cr.yp.to/papers.html#ed25519

[ED25519-B-REF]:

https://tools.ietf.org/html/draft-josefsson-eddsa-ed25519-03#section-5:

[PRNG-REFS]:

http://projectbullrun.org/dual-ec/ext-rand.html

https://lists.torproject.org/pipermail/tor-dev/2015-November/009954.html

[SRV-TP-REFS]:

https://lists.torproject.org/pipermail/tor-dev/2016-April/010759.html

[VANITY-REFS]:

https://github.com/Yawning/horse25519

[ONIONADDRESS-REFS]:

https://lists.torproject.org/pipermail/tor-dev/2017-January/011816.html

[TORSION-REFS]:

https://lists.torproject.org/pipermail/tor-dev/2017-April/012164.html

https://getmonero.org/2017/05/17/disclosure-of-a-major-bug-in-cryptonote-based-currencies.html

Appendix A. Signature scheme with key blinding [KEYBLIND]

A.1. Key derivation overview

As described in [IMD:DIST] and [SUBCRED] above, we require a "key

blinding" system that works (roughly) as follows:

There is a master keypair (sk, pk).

Given the keypair and a nonce n, there is a derivation function

that gives a new blinded keypair (sk\_n, pk\_n). This keypair can

be used for signing.

Given only the public key and the nonce, there is a function

that gives pk\_n.

Without knowing pk, it is not possible to derive pk\_n; without

knowing sk, it is not possible to derive sk\_n.

It's possible to check that a signature was made with sk\_n while

knowing only pk\_n.

Someone who sees a large number of blinded public keys and

signatures made using those public keys can't tell which

signatures and which blinded keys were derived from the same

master keypair.

You can't forge signatures.

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

A.2. Tor's key derivation scheme

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

(This is an ECC group, so remember that scalar multiplication is the

trapdoor function, and it's defined in terms of iterated point

addition. See the Ed25519 paper [Reference ED25519-REFS] for a fairly

clear writeup.)

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

B = (15112221349535400772501151409588531511454012693041857206046113283949847762202,

46316835694926478169428394003475163141307993866256225615783033603165251855960)

Assume B has prime order l, so lB=0. Let a master keypair be written as

(a,A), where a is the private key and A is the public key (A=aB).

To derive the key for a nonce N and an optional secret s, compute the

blinding factor like this:

h = H(BLIND\_STRING | A | s | B | N)

BLIND\_STRING = "Derive temporary signing key" | INT\_1(0)

N = "key-blind" | INT\_8(period-number) | INT\_8(period\_length)

B = "(1511[...]2202, 4631[...]5960)"

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

h[0] &= 248;

h[31] &= 63;

h[31] |= 64;

and do the key derivation as follows:

private key for the period:

a' = h a mod l

RH' = SHA-512(RH\_BLIND\_STRING | RH)[:32]

RH\_BLIND\_STRING = "Derive temporary signing key hash input"

public key for the period:

A' = h A = (ha)B

Generating a signature of M: given a deterministic random-looking r

(see EdDSA paper), take R=rB, S=r+hash(R,A',M)ah mod l. Send signature

(R,S) and public key A'.

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

(If the signature is valid,

SB = (r + hash(R,A',M)ah)B

= rB + (hash(R,A',M)ah)B

= R + hash(R,A',M)A' )

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

See [KEYBLIND-REFS] for an extensive discussion on this scheme and

possible alternatives. Also, see [KEYBLIND-PROOF] for a security

proof of this scheme.

Appendix B. Selecting nodes [PICKNODES]

Picking introduction points

Picking rendezvous points

Building paths

Reusing circuits

(TODO: This needs a writeup)

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

EDITORIAL NOTE: The author thinks that it's silly to brute-force the

keyspace for a key that, when base-32 encoded, spells out the name of

your website. It also feels a bit dangerous to me. If you train your

users to connect to

llamanymityx4fi3l6x2gyzmtmgxjyqyorj9qsb5r543izcwymle.onion

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

connecting to

llamanymityb4sqi0ta0tsw6uovyhwlezkcrmczeuzdvfauuemle.onion

Nevertheless, people are probably going to try to do this, so here's a

decent algorithm to use.

To search for a public key with some criterion X:

Generate a random (sk,pk) pair.

While pk does not satisfy X:

Add the number 8 to sk

Add the point 8\*B to pk

Return sk, pk.

We add 8 and 8\*B, rather than 1 and B, so that sk is always a valid

Curve25519 private key, with the lowest 3 bits equal to 0.

This algorithm is safe [source: djb, personal communication] [TODO:

Make sure I understood correctly!] so long as only the final (sk,pk)

pair is used, and all previous values are discarded.

To parallelize this algorithm, start with an independent (sk,pk) pair

generated for each independent thread, and let each search proceed

independently.

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

search scheme.

Appendix D. Numeric values reserved in this document

[TODO: collect all the lists of commands and values mentioned above]

Appendix E. Reserved numbers

We reserve these certificate type values for Ed25519 certificates:

[08] short-term descriptor signing key, signed with blinded

public key. (Section 2.4)

[09] intro point authentication key, cross-certifying the descriptor

signing key. (Section 2.5)

[0B] ed25519 key derived from the curve25519 intro point encryption key,

cross-certifying the descriptor signing key. (Section 2.5)

Note: The value "0A" is skipped because it's reserved for the onion key

cross-certifying ntor identity key from proposal 228.

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

This appendix section specifies the contents of the HiddenServiceDir directory:

- "hostname" [FILE]

This file contains the onion address of the onion service.

- "private\_key\_ed25519" [FILE]

This file contains the private master ed25519 key of the onion service.

[TODO: Offline keys]

- "./authorized\_clients/" [DIRECTORY]

"./authorized\_clients/alice.auth" [FILE]

"./authorized\_clients/bob.auth" [FILE]

"./authorized\_clients/charlie.auth" [FILE]

If client authorization is enabled, this directory MUST contain a ".auth"

file for each authorized client. Each such file contains the public key of

the respective client. The files are transmitted to the service operator by

the client.

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

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

Appendix G. Managing authorized client data [CLIENT-AUTH-MGMT]

Hidden services and clients can configure their authorized client data either

using the torrc, or using the control port. This section presents a suggested

scheme for configuring client authorization. Please see appendix

[HIDSERVDIR-FORMAT] for more information about relevant hidden service files.

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

G.1. Configuring client authorization using torrc

G.1.1. Hidden Service side configuration

A hidden service that wants to enable client authorization, needs to

populate the "authorized\_clients/" directory of its HiddenServiceDir

directory with the ".auth" files of its authorized clients.

When Tor starts up with a configured onion service, Tor checks its

<HiddenServiceDir>/authorized\_clients/ directory for ".auth" files, and if

any recognized and parseable such files are found, then client

authorization becomes activated for that service.

G.1.2. Service-side bookkeeping

This section contains more details on how onion services should be keeping

track of their client ".auth" files.

For the "descriptor" authentication type, the ".auth" file MUST contain

the x25519 public key of that client. Here is a suggested file format:

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

Here is an an example:

descriptor:x25519:OM7TGIVRYMY6PFX6GAC6ATRTA5U6WW6U7A4ZNHQDI6OVL52XVV2Q

Tor SHOULD ignore lines it does not recognize.

Tor SHOULD ignore files that don't use the ".auth" suffix.

G.1.3. Client side configuration

A client who wants to register client authorization data for onion

services needs to add the following line to their torrc to indicate the

directory which hosts ".auth\_private" files containing client-side

credentials for onion services:

ClientOnionAuthDir <DIR>

The <DIR> contains a file with the suffix ".auth\_private" for each onion

service the client is authorized with. Tor should scan the directory for

".auth\_private" files to find which onion services require client

authorization from this client.

For the "descriptor" auth-type, a ".auth\_private" file contains the

private x25519 key:

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

The keypair used for client authorization is created by a third party tool

for which the public key needs to be transferred to the service operator

in a secure out-of-band way. The third party tool SHOULD add appropriate

headers to the private key file to ensure that users won't accidentally

give out their private key.

G.2. Configuring client authorization using the control port

G.2.1. Service side

A hidden service also has the option to configure authorized clients

using the control port. The idea is that hidden service operators can use

controller utilities that manage their access control instead of using

the filesystem to register client keys.

Specifically, we require a new control port command ADD\_ONION\_CLIENT\_AUTH

which is able to register x25519/ed25519 public keys tied to a specific

authorized client.

[XXX figure out control port command format]

Hidden services who use the control port interface for client auth need

to perform their own key management.

G.2.2. Client side

There should also be a control port interface for clients to register

authorization data for hidden services without having to use the

torrc. It should allow both generation of client authorization private

keys, and also to import client authorization data provided by a hidden

service

This way, Tor Browser can present "Generate client auth keys" and "Import

client auth keys" dialogs to users when they try to visit a hidden service

that is protected by client authorization.

Specifically, we require two new control port commands:

IMPORT\_ONION\_CLIENT\_AUTH\_DATA

GENERATE\_ONION\_CLIENT\_AUTH\_DATA

which import and generate client authorization data respectively.

[XXX how does key management work here?]

[XXX what happens when people use both the control port interface and the

filesystem interface?]

Appendix F. Two methods for managing revision counters.

Implementations MAY generate revision counters in any way they please,

so long as they are monotonically increasing over the lifetime of each

blinded public key. But to avoid fingerprinting, implementors SHOULD

choose a strategy also used by other Tor implementations. Here we

describe two, and additionally list some strategies that implementors

should NOT use.

F.1. Increment-on-generation

This is the simplest strategy, and the one used by Tor through at

least version 0.3.4.0-alpha.

Whenever using a new blinded key, the service records the

highest revision counter it has used with that key. When generating

a descriptor, the service uses the smallest non-negative number

higher than any number it has already used.

In other words, the revision counters under this system start fresh

with each blinded key as 0, 1, 2, 3, and so on.

F.2. Encrypted time in period

This scheme is what we recommend for situations when multiple

service instances need to coordinate their revision counters,

without an actual coordination mechanism.

Let T be the number of seconds that have elapsed since the descriptor

became valid, plus 1. (T must be at least 1.) Implementations can use the

number of seconds since the start time of the shared random protocol run

that corresponds to this descriptor.

Let S be a secret that all the service providers share. For

example, it could be the private signing key corresponding to the

current blinded key.

Let K be an AES-256 key, generated as

K = H("rev-counter-generation" | S)

Use K, and AES in counter mode with IV=0, to generate a stream of T

\* 2 bytes. Consider these bytes as a sequence of T 16-bit

little-endian words. Add these words.

Let the sum of these words be the revision counter.

Cryptowiki attributes roughly this scheme to G. Bebek in:

G. Bebek. Anti-tamper database research: Inference control

techniques. Technical Report EECS 433 Final Report, Case

Western Reserve University, November 2002.

Although we believe it is suitable for use in this application, it

is not a perfect order-preserving encryption algorithm (and all

order-preserving encryption has weaknesses). Please think twice

before using it for anything else.

(This scheme can be optimized pretty easily by caching the encryption of

X\*1, X\*2, X\*3, etc for some well chosen X.)

For a slow reference implementation, see src/test/ope\_ref.py in the

Tor source repository. [XXXX for now, see the same file in Nick's

"ope\_hax" branch -- it isn't merged yet.]

This scheme is not currently implemented in Tor.

F.X. Some revision-counter strategies to avoid

Though it might be tempting, implementations SHOULD NOT use the

current time or the current time within the period directly as their

revision counter -- doing so leaks their view of the current time,

which can be used to link the onion service to other services run on

the same host.

Similarly, implementations SHOULD NOT let the revision counter

increase forever without resetting it -- doing so links the service

across changes in the blinded public key.