Obfs2

Pluggable Transport documentation series.[*]

Carolin Zöbelein[†]
Independent mathematical scientist
Josephsplatz 8, 90403 Nürnberg, Germany

ABSTRACT

STATUS: Draft

KEYWORDS

Tor, Bridge, Scary, Obscuration, Censorship, Circumvention, Pluggable Transport

PREAMBLE

This paper is part of a paper documentation series about Pluggable Transports (PTs) [3], how they work and their strengths and weaknesses.

1 INTRODUCTION

During the history of digital networks, we have been confronted more and more with the phenomenon of internet censorship and blocking by governments [4]. So, over the years, more and more circumvention tools were developed and also have been blocked due to deep packet inspections and the detailed analysis of its content. This lead us to, so-called, *Pluggable Transports (PTs)* [3], which help to bypass censorship attemps by transforming the traffic between client and server, in such a ways that it looks like innocent traffic.

In this paper, we will talk about *obfs2*, a protocol obfuscation layer for TCP protocols. We will show how it works, what we can do with it and which strengths and weaknesses it has.

1.1 Outline

TODO

2 OBFS2

Obfs2 is a protocol obfuscation layer for TCP protocols, to keep a third party from telling what protocol is in use based on message contents [2]. It's the continuation of brl's ssh obfuscation protocol [2] [1].

2.1 Overview

The protocol consists of two phases.

- First: The parties establish keys
- Second: The parties exchange superenciphered traffic.

[*]The author believes in the importance of the independence of research and is funded by the public community. If you also believe in this values, you can find ways for supporting the author's work here: https://research.carolin-zoebelein.de/crowdfunding.html
[†]https://research.carolin-zoebelein.de, *E-mail address:* contact@carolin-zoebelein.de, PGP: D4A7 35E8 D47F 801F 2CF6 2BA7 927A FD3C DE47 E13B

October 29, 2018.

2.2 Notation

Given are two parties: the 'initiator' (INIT), which opens the connection and can mostly be associated with a client, and the 'responder' (RESP), which accepts the connection and can mostly be associated with a server.

We use the following primitives,

- H(x) is SHA256 of x
- $H^n(x)$ is H(x) called iteratively n times
- Enc(K,s) is the AES-CTR-128 encryption of s using K as key

notation

- x | y is the concatenation of x and y
- UINT32(n) is the 4 byte value of n in big-endian (network) order
- SR(n) is n bytes of strong random data
- WR(n) is n bytes of weaker random data
- "xyz" is the ASCII characters 'x', 'y', and 'z', not NUL-terminated
- s[:n] is the first n bytes of s
- s[n:] is the last n bytes of s

and constants

- MAGIC_VALUE = 0x2BF5CA7E
- SEED_LENGTH = 16
- MAX_PADDING = 8192
- HASH_ITERATIONS = 100000

as well as

- KEYLEN = 16 is the length of the key used by Enc(K,s)
- IVLEN = 16 is the length of the IV used by Enc(K,s)
- HASHLEN = 32 is the length of the output of H()
- MAC(s, x) = H(s | x | s)

according to [2]. A "byte" is an 8-bit octet and we require that HASHLEN >= KEYLEN + IVLEN.

2.3 The key establishment phase

The key establishment phase consists of several substeps.

2.3.1 The given values. Given are the constants MAGIC_VALUE, SEED_LENGTH, MAX_PADDING and HASH_ITERATIONS and the lengths KEYLEN, IVLEN and HASHLEN.

2.3.2 Generating initial values. The 'initiator' (see listing 1) generate a seed, a padding key and a random PADLEN in range from 0 through MAX_PADDING (inclusive).

Listing 1: Generate INIT seed and padding key [2].

```
    INIT_SEED  =  SR(SEED_LENGTH)
    INIT_PAD_KEY  =  MAC("Initiator obfuscation padding
        ", INIT_SEED)[:KEYLEN]
    PADLEN  =  R([0:MAX_PADDING])
```

The 'responder' (see listing 2) do it in the same way.

Listing 2: Generate RESP seed and padding key [2].

```
RESP_SEED = SR(SEED_LENGTH)
RESP_PAD_KEY = MAC("Responder obfuscation padding
    ", RESP_SEED)[:KEYLEN]
PADLEN = R([0:MAX_PADDING])
```

2.3.3 Init messages. After generating the initial values [2], the initiator (see listing 3) sends

Listing 3: INIT's init message [2].

```
INIT_SEED | Enc(INIT_PAD_KEY, UINT32(MAGIC_VALUE)
    | UINT32(PADLEN) | WR(PADLEN))
```

encrypted by the key information given by the SEED, to the responder (see listing 4), which do it likewise.

Listing 4: RESP's init message [2].

```
RESP_SEED | Enc(RESP_PAD_KEY, UINT32(MAGIC_VALUE)
    | UINT32(PADLEN) | WR(PADLEN))
```

After receiving the SEED from the other party, each party decrypts the other party's padding key value and the next 8 bytes.

It checks the MAGIC_VALUE and the PADLEN value for conclusiveness and close the connection immediately in case of invalidity, otherwise, it read the remaining PADLEN bytes of padding data and discard them.

2.3.4 Final key initialisations. If the messages above are valid, the parties derive additional keys (see listing 5).

Listing 5: Final key initialisations [2].

```
INIT_SECRET = MAC("Initiator obfuscated data",
    INIT_SEED|RESP_SEED)
RESP_SECRET = MAC("Responder obfuscated data",
    INIT_SEED|RESP_SEED)

INIT_KEY = INIT_SECRET[:KEYLEN]
INIT_IV = INIT_SECRET[KEYLEN:]

RESP_KEY = RESP_SECRET[:KEYLEN]
RESP_IV = RESP_SECRET[KEYLEN:]
```

The INIT_KEY value keys a stream cipher used to encrypt values from initiator to responder and the stream cipher's IV is INIT_IV. The responder do it in the same way for it's values.

3 CONCLUSIONS

A APPENDIX

REFERENCES

[1] https://github.com/brl/obfuscated-openssh. 2009. GitHub: Handshake Obfuscation. (2009).

[2] https://gitweb.torproject.org/pluggable-transports/obfsproxy.git/tree/doc/obfs2/obfs2-protocol-spec.txt. 2015. Obfs2 specification. (2015).

[3] https://www.torproject.org/docs/pluggable-transports.html.en. 2018. Tor: Pluggable Transports. (2018).

[4] Philipp Winter and Stefan Lindskog. 2012. How the Great Firewall of China is blocking Tor. In *Proceedings of the USENIX Workshop on Free and Open Communications on the Internet (FOCI 2012).*

LICENSE