TLSFilter: An Application-Level Firewall for Transport Layer Security

Final Report

Mat Phillips
mp2509@doc.ic.ac.uk

June 16, 2014

Abstract

The proliferation of misconfigured and vulnerable SSL/TLS implementations has led to a situation where a substantial proportion of encrypted traffic can no longer be considered secure. The 2013/14 publication of government signals intelligence capabilities and security vendor coercion underline the realistic potential for current and future vulnerabilities to be exploited on a mass scale.

This report documents the creation and assessment of an application-level firewall that ensures all TLS connections meet a minimum level of security, irrespective of the underlying TLS software implementations. As security is enforced at the network layer, it can be used to secure an entire network of IP devices including embedded systems and network appliances that often suffer from a lack of, or infrequent, security updates.

The firewall combines techniques for mitigating a variety of TLS architectural weaknesses, including man-in-the-middle attacks, as well as providing a general-purpose platform for future extension. Firewall rules can be grouped into profiles and applied on the basis of IP address, allowing the fulfilment of varied use cases. By using TCP reassembly logic from the Linux kernel and by performing deep packet inspection on all network traffic, it is immune to a class of fragmentation and obfuscation-based techniques used to evade traditional network filtering software.

The analysis shows that use of the firewall results in a quantifiable net security gain for all nine scenarios tested under a modified Schneier attack tree model. The additional latency introduced is in the negligible order of tens of milliseconds; single-connection throughput performance is reduced by 0.8%; and concurrent throughput is reduced by 25% at 200 concurrent connections. The analysis of approximately 460,000 websites led to the creation of a default configuration that improves security at the expense of reducing reachability by 5.5%.

Acknowledgements

I would like to thank my supervisor, Dr. Jeremy Bradley, for his boundless enthusiasm, his willingness to supervise this project and for guiding me in the right direction on numerous occasions. Thanks also to Ralph Oppliger for writing *SSL and TLS: Theory and Practice*, which saved me from hours of RFC torture and Nitin Nihalani for checking my unexpected performance results and independently verifying the sanity of my methodology.

Contents

List of Figures

Chapter 1

Introduction

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) have become the standard protocols for securing information exchange over untrusted networks and we have become increasingly reliant on them as a greater proportion of our communication occurs over the internet. The latest version of TLS is highly extensible, and was built under the presumption that it would become the secure basis upon which future—more application-specific—protocols are built.

As our reliance on these protocols increases, so too do discoveries of significant vulnerabilities in implementations and weaknesses in configurations of TLS libraries. Though software updates are often quickly released to correct these vulnerabilities, many publicly-reachable TLS endpoints remain vulnerable. The discovery of two serious implementation vulnerabilities, Heartbleed (Section 2.4.5) and ChangeCipherSpec injection (Section 2.4.5), in mid-2014 evidences the scale and importance of this problem.

In addition, the capabilities of adversaries have, until recently, been underestimated, making existing threat models less relevant and the software designed around them less suited to the current environment. Public knowledge of global, well-resourced attackers such as the UK's GCHQ and the US National Security Agency as well as the complicity of software vendors to adopt defective security standards underlines the importance of problems with TLS and the realistic potential for existing (and future) weaknesses to be exploited.

My objective is to produce software that mitigates numerous technical threats to TLS, bearing in mind the realities of existing software deployments and the likelihood of new attacks executed by these well-resourced adversaries. Moreover, it is my objective to build software that provides a *measurable* increase in security and interoperates with, rather than replaces, current systems.

I claim to have realised this objective through the creation of TLSFilter: a stateful, application-level firewall that blocks TLS connections that do not meet a minimum standard of security. My intention is that the core rule set can be updated in line with the discovery and publication of new weaknesses, allowing the protection of entire networks of client and server software without having

to wait for the creation and manual roll-out of security updates.

TLSFilter can be configured to block specific vulnerabilities including Heartbleed (Section 2.4.5) and ChangeCipherSpec injection (Section 2.4.5). Furthermore, its logic can be extended to an arbitrary level of complexity through the plugin API and, combined with the granular application of rule sets over IP addresses, TLSFilter allows the specification of any level of security.

My evaluation (Chapter 6) seeks to assess the usefulness of TLSFilter in terms of the trade-off between security and network reachability through analysis of the TLS configurations of the most popular 460,000 websites. This section also quantitatively analyses the gain in security using a modification of the Schneier attack tree modelling formalism and includes an assessment of the performance in terms of throughput, latency and concurrency.

1.1 Project Objectives

The broad objectives of my project are:

- To prevent insecure TLS connections, both inbound and outbound, network-wide
- To permit the specification of disparate security policies on the basis of destination IP address or network
- To facilitate the rapid response to widespread TLS vulnerabilities
- To provide a general-purpose TLS firewall engine that can be easily extended
- To mitigate packet fragmentation and protocol obfuscation evasion techniques

Chapter 2

Background

This chapter contains an overview of the goals of SSL/TLS, details of the underlying protocol messages, background on the cryptography required and specifics of the most important threats to the protocol. To be accessible to the lay reader very little prior knowledge is assumed so those who are already fully aware of the topics discussed may wish to skip many of the sections in this chapter.

2.1 SSL/TLS Protocol Introduction

Secure Sockets Layer (SSL) and, its successor, Transport Layer Security (TLS) are suites of protocols used to secure transport-layer communication. (Often both suites are referred to as SSL or TLS, since TLS is simply an extension of SSL.) TLS aims to provide three main properties—authenticity, confidentiality and integrity—and does so through both symmetric and public key cryptography.

2.1.1 Authenticity

Authentication is the process that verifies that clients and servers are who they purport to be. In TLS, authentication takes place outside of the scope of the protocol and relies on external public key infrastructure (PKI).

Prior to the configuration of a TLS server, an X.509 certificate containing identity information and public key(s) must be generated [1]. This certificate must then be digitally signed by a certificate authority (CA) who is a third party trusted to validate the details of certificates. The resulting signed certificate can then be sent to clients as part of the protocol handshake, where the client will independently verify that the certificate has been signed by a CA they trust.

The reverse process for client authentication is also supported by TLS. See Section 2.1.4 for details of this functionality in greater depth.

2.1.2 Confidentiality

Confidentiality is a property of a system that ensures message content is only readable by the intended recipients. In TLS, this is implemented using symmetric cryptography—for example, using the Rijndael cipher that is the basis of the Advanced Encryption Standard (AES)—in conjunction with a key (or secret) known only by the client and server.

Key Agreement

In the majority of applications a client and server will not possess a pre-shared key (PSK), so a key exchange algorithm must be used to share a key across an untrusted communications channel. Currently, the key exchange algorithms supported by TLS use public-key cryptography based upon the integer factorization problem (e.g., Rivest-Shamir-Adleman (RSA)) and the discrete logarithm problem (e.g., Diffie-Hellman).

The RSA cryptosystem can be used to securely exchange a key from a client Alice to a server Bob. Alice randomly generates a key, encrypts it under the public key in Bob's certificate, sends this over the untrusted communications infrastructure to Bob, who decrypts the message with his private key.

Diffie-Hellman requires Alice and Bob agree to a public value g and large public prime number p. Alice chooses a secret value x and sends $g^x \mod p$ to Bob; Bob chooses a secret value y and sends $g^y \mod p$ to Alice. Alice and Bob both multiply the values received with the values they sent, and this is the shared key value. This works because $(g^x \mod p)(g^y \mod p) = g^{xy} \mod p = g^{yx} \mod p = (g^y \mod p)(g^x \mod p)$. Diffie-Hellman is broadly considered to be a better choice than RSA because both parties contribute entropy to the key.

Symmetric Ciphers

Prior to the most recent version, the two classes of symmetric ciphers available in TLS were block ciphers and stream ciphers.

Stream ciphers are the simpler of the two, and allow the encryption of arbitrary-length plaintexts. Stream ciphers work by deterministically generating a pseudorandom stream of data from a key, which is known as a keystream and is combined with the plaintext to yield the ciphertext. Generally stream ciphers use the exclusive-or operator to perform this function, as shown in Figure 2.1. This method of encrypting data is generally considered to be computationally efficient, so mandatory support for stream ciphers such as RC4 has been required by every version of the TLS protocol.

Block ciphers operate on fixed-size blocks of data and consequently plaintext must be padded to fill a multiple of the block size. In TLS, block ciphers are used in cipher block chaining (CBC) mode, where the nth plaintext block is XORed with the ciphertext of the $n-1$th block before being encrypted under the block cipher with the key (Figure 2.2). This means an initialisation vector (IV) is required to encrypt the first block and the encryption under this mode is not parallelisable,

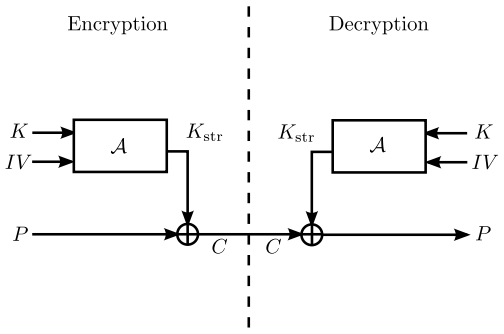


Figure 2.1: Stream cipher operation

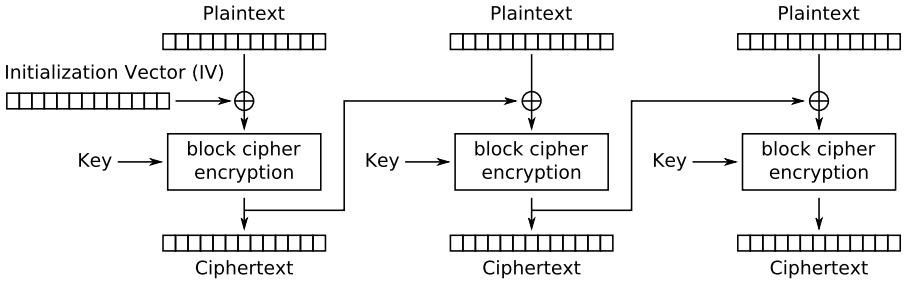


Figure 2.2: Encryption under Cipher Block Chaining (CBC) mode

so block ciphers in CBC are considered less efficient than stream ciphers.

TLS v1.2 adds support for Authenticated Encryption with Associated Data (AEAD)[2] ciphers by introducing cipher suites containing block ciphers in Galois/Counter Mode (GCM). In general AEAD modes allow the encryption *and* authentication of plaintext data (PDATA) to be a single process, rather than more traditional approaches where a distinct authentication process is executed after (or before) encryption. It also allows for additional data (ADATA) to be authenticated but not encrypted.

GCM uses a modified form of encryption in counter (CTR) mode[3]. CTR mode uses a block cipher (like Rijndael/AES) as a stream cipher (like RC4); the keystream is generated by encrypting a block consisting of a nonce appended with a counter that is initialised to zero. Exclusive-or is then used to combine the plaintext with this encrypted block, as in Figure 2.3. To generate more keystream, the counter field is simply incremented and the process is otherwise identical. Useful features of this mode of operation is that the decryption function is identical to the encryption function (and parallelisable), and that no padding of plaintext is required (unlike CBC).

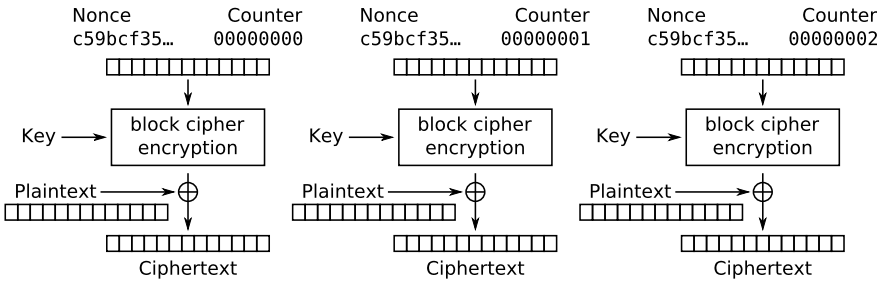


Figure 2.3: Encryption under counter (CTR) mode

The output of GCM is referred to as a tag, which is authenticated and may contain only plaintext (used to produce a MAC), only ciphertext, or both. Tags produced are initially equal to the block length of the cipher used (to a minimum of 128 bits), though can be truncated without issue. Tags are produced by encrypting the result of the $GHASH$ function defined in GCM's specification[4], which uses a modified CTR mode.

2.1.3 Integrity

Message integrity ensures that modifications to messages by third parties can be detected. This is implemented using message authentication codes (MACs) derived from cryptographic hash functions, where the contents of a message is hashed and signed by the sender to verify that the message has not been altered in transit.

In recent versions of TLS, strong hash functions—i.e., those where collisions are computationally difficult to find—are used in conjunction with a keyed-hash message authentication code (HMAC), to simultaneously provide message integrity and authentication.

As TLS aims to be a future-proof protocol, the sets of algorithms that implement the cryptographic primitives mentioned above (known as cipher suites) are fully configurable and are standardised by the Internet Assigned Numbers Authority (IANA).

2.1.4 SSL/TLS Connection Handshake

Figure 2.4 shows the anatomy of the handshake for the TLS v1.2 protocol [2], where optional messages are shown with dashed arrows. The handshake protocol works as follows:

1. `ClientHello` - this message initiates a connection from the client and contains the highest SSL/TLS version supported, a session ID, a list of cipher suites and compression methods. As of TLS v1.2, the contents of this message may include a list of TLS feature set extensions supported by the client.

2. `ServerHello` - the server responds with a similar message, specifying the cipher suite chosen for communication, and may list the subset of the TLS extensions it supports.

3. `SupplementalData` *optional* - if the client supports an appropriate TLS extension, the server may send arbitrary application-specific data.

4. `Certificate` *optional* - the server may send an X.509 identity certificate to the client.

5. `ServerKeyExchange` *optional* - if the cipher suite specifies a key exchange algorithm that requires server interaction, such as Diffie-Hellman, the server sends the data in this message.

6. `CertificateRequest` *optional* - if the server requires mutual authentication, this message is sent to request an X.509 certificate from the client.

7. `ServerHelloDone` - this message signals the end of the server's half of the handshake.

8. `SupplementalData` *optional* - the client may then send arbitrary data to the server for application-specific purposes.

9. `Certificate` *optional* - if a `CertificateRequest` was received, the client will send an X.509 identity certificate to the server.

10. `ClientKeyExchange` - the client will either initiate (in the case of RSA) or complete (in the case of Diffie-Hellman) a key exchange with this message.

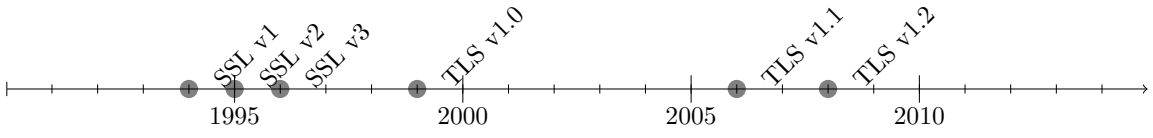
11. `CertificateVerify` *optional* - the client will verify that the server possesses the private key associated with the public key from its X.509 certificate.

12. `ChangeCipherSpec` - this message signals that all further messages from the client to the server will be encrypted under the shared secret.

13. `Finished` - this message contains a hash of all previous messages received, to ensure that the handshake was not manipulated by a third party.

14. Subsequent `ChangeCipherSpec` and `Finished` messages are then sent from the server to the client. If the plaintext of the `Finished` messages match the locally-generated hashes, the handshake is complete and encrypted application data can be exchanged.

2.2 SSL/TLS Version History



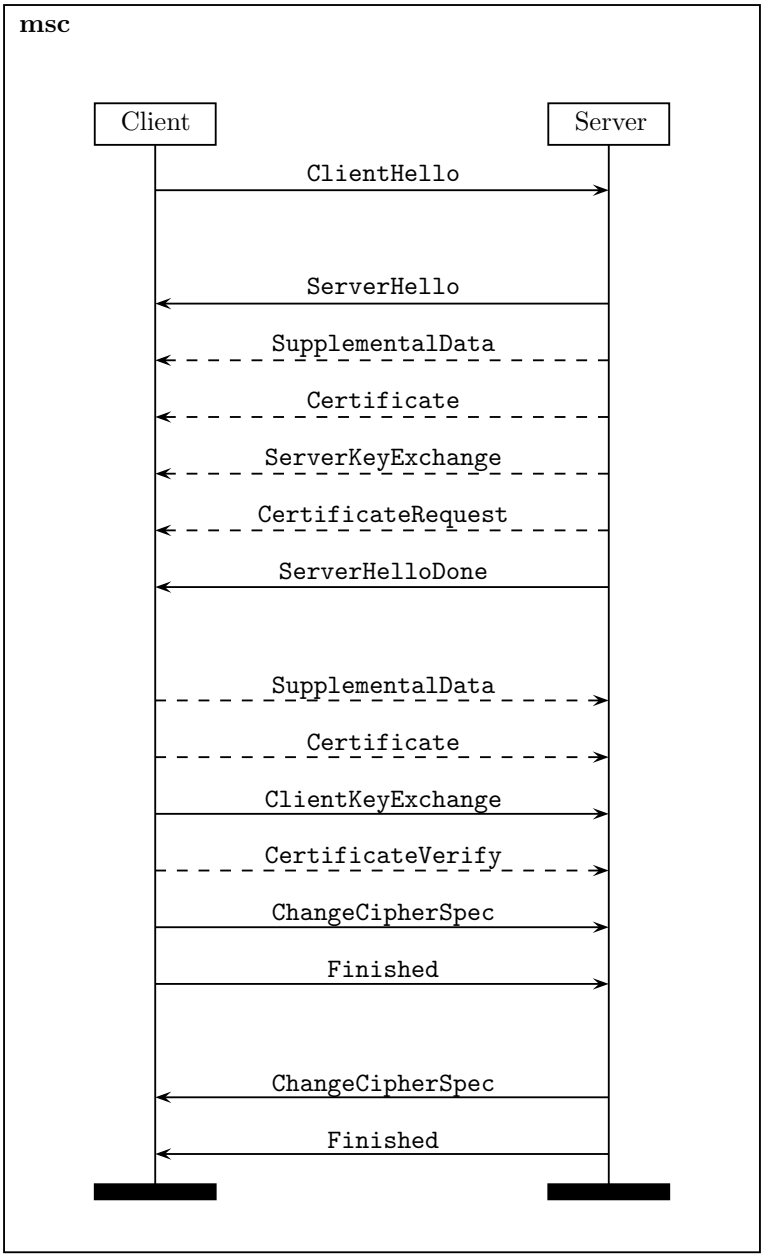


Figure 2.4: Message sequence chart for a TLSv1.2 handshake

SSL v1

The first version of SSL was created by Netscape and was not released as a public protocol.

Initially, it did not feature message sequence numbers nor were efforts made to ensure message integrity, so it was vulnerable to message-replay attacks and undetectable message modification. Message sequence numbers were later added and cyclic redundancy checksumming (CRC) used in a failed attempt to mitigate these weaknesses.

SSL v2

Version two of the SSL protocol was the first version to be published as a draft internet standard. It was built on top of Netscape's SSL v1, published in 1995 and patented but released into the public domain soon after.

The cryptographic hash function MD5 replaced CRC for message integrity, as MD5 was considered to be collision-resistant at the time. In addition, certificate support was added to the protocol handshake for client and server authentication, but certificates were required to be signed directly by a root certificate authority to be valid.

Various weaknesses mean that SSL v2 is no longer in use.

SSL v3

SSL v3 is the version with the most significant protocol changes from its predecessor, and is the most widely-adopted version[8]. Despite being released in 1996[7], it is still in use today and remains the oldest version of the protocol supported by modern SSL/TLS implementations.

Message authentication code generation was modified to use both MD5 and SHA1 hash algorithms simultaneously[7], to mitigate concerns that collision weaknesses discovered in MD5 could compromise security. It was also hoped that relying on two different hashing functions would mean that SSL continued to provide message integrity in the event that serious vulnerabilities were discovered in either algorithm.

Authentication in this version no longer required certificates to be signed directly by a root CA, and introduced support for intermediate CAs. These are certification authorities whom a root CA has entrusted with the ability to sign certificates that carry the same validity as a certificate signed directly by the root CA. A certificate is accepted by a client if there exists an arbitrary-length chain of trust from a root CA, via zero or more intermediate CAs, to the server's certificate.

SSL v3 also mandated the use of different keys for encryption and authentication to reduce the potential impact of vulnerable cipher and MAC algorithm combinations[7]. For example, if an SSL connection uses a common key for confidentiality under AES-CBC and message integrity under AES-CBC-MAC, then a vulnerability in the MAC algorithm that leads to the retrieval of key material could also break the confidentiality of the message. In TLS v1.2, an exception to this

practice was made for Galois/Counter mode (GCM)[2], which intrinsically supports the secure use of a single key for encryption and authentication.

Compression support was tentatively added to the protocol to reduce the increase in handshake overhead that these changes brought[7]. However, supported compression algorithms were not defined until later revisions of the protocol.

TLS v1.0

TLS v1.0 was released in 1999[6] and, despite the change of name from SSL to TLS, does not represent changes as significant as those from SSL v2 to SSL v3. Accordingly, it is referred to in the protocol handshake as version "3.1", and this tradition has continued for subsequent versions of TLS.

Message authentication is standardised by enforcing the use of a fixed keyed-hash message authentication code (HMAC) function, dropping support for the previous non-keyed MD5-SHA1 concatenation MAC. $HMAC(K, m)$ is defined to be $H((K \oplus opad)||H((K \oplus ipad)||m))$, where K is a key, m is the message to be authenticated, H is a cryptographic hash function, *opad* is the 'outer padding' consisting of `0x5c` repeated for one block, *ipad* is the 'inner padding' consisting of `0x36` repeated for one block, and || denotes concatenation.

A single algorithm, specified in the cipher suite chosen by the server in the handshake, is now used to provide message integrity and authentication[6]. This reduces the reliance on particular cryptographic hash functions and means that vulnerabilities in algorithms do not necessitate changes in the protocol specification.

To mitigate traffic analysis whereby an attacker deduces the encrypted payload by its size, the protocol now permits up to 255 bytes of padding. In SSL v3, padding was only permitted under CBC mode to extend the plaintext to a multiple of the block size[7], rather than as a security concern.

Changes were also made to the cipher suites supported. U.S. cryptographic export regulations were about to be relaxed when the TLS v1.0 specification was being written, so triple-DES (3DES) replaced DES as the block cipher that required mandatory support and FORTEZZA—a cipher developed for the U.S. Government's ill-fated Clipper chip project—was removed[6]. In 2006, elliptic curve cipher suites were added to the TLS v1.0 specification retroactively[9].

TLS v1.1

TLS v1.1 was released in 2006 and represents minor fixes to issues with TLS v1.0[5].

CBC mode padding attacks were mitigated (Section 2.4.2), and the number of valid SSL alert message classes was decreased to reduce the information available to an adversary.

Explicit random per-message IVs are used for connections that negotiate a block cipher in CBC mode. In SSL v3 and TLS v1.0, an explicit initial IV is sent as part of the handshake to be used for the first message and IVs for subsequent messages are derived implicitly from the ciphertext of

the preceding message[7, 6], which allows an adversary to predict IVs. It was later discovered in 2011 that this weakness led to an attack known as BEAST (Section 2.4.3).

TLS v1.1 also modernises cipher suite support; suites containing "export-grade" ciphers—weak ciphers with short key lengths permitted to be exported under old U.S. regulations—are no longer allowed to be chosen by a server as part of a TLS v1.1 connection handshake[5].

TLS v1.2

Released in 2008, TLS v1.2[2] is the latest and most secure version of the protocol but has yet to gain widespread adoption. According to the Trustworthy Internet Movement, as of January 2014, TLS v1.2 is supported by 25.7% of the top 200,000 most trafficked HTTPS-enabled web sites[8], in contrast to SSL v3.0 which is supported by 99.5%.

The supported cipher suite list was revised to remove those containing weak ciphers including DES (a cipher that required mandatory support in previous versions) and IDEA. The 256-bit variant of the secure hash algorithm (SHA-256) was added to various suites to replace the older, and weaker, MD5 and SHA1 algorithms[2] .

Support was also added for authenticated encryption with additional data (AEAD) cipher suites, notably 128 and 256-bit AES in Galois/Counter mode, which is generally considered to be the most secure cipher suite available. (AES-GCM in particular offers significant performance benefits on Intel hardware that supports AES-NI or the `PCLMULQDQ` instruction.)

A client and server can now be configured to use pre-shared keys (PSKs), as opposed to requiring a Diffie-Hellman or RSA key exchange as part of the connection handshake. This functionality is intended for situations where public key operations are too computationally expensive, or otherwise unsuitable.

DEFLATE compression was also added in TLS v1.2, making it the only option to exploit the protocol support for compression introduced in SSL v3[2], but this led to an attack known as CRIME (Section 2.4.3).

2.2.1 TLS Extensions

TLS extensions were retroactively added to all versions of TLS, and are intended to add flexibility to the protocol by broadening its possible use cases. In doing so, the protocol authors hope to discourage the creation of single-purpose security protocols and their libraries, which the TLS v1.2 RFC claims may risk the introduction of possible new weaknesses[2].

2.3 Miscellaneous TLS Features

2.3.1 Ephemeral Keys

Ephemeral keys are temporary keys that are discarded permanently after session cessation.

Forward Secrecy

Forward secrecy is a property of a cryptosystem that uses ephemeral keys to ensure that, if long-term keying material is compromised, the confidentiality of individual sessions is still maintained. For TLS, forward secrecy is defined by the establishment a single ephemeral key per session, and the classes of cipher suites containing key exchange algorithms that support this carry the DHE (Diffie-Hellman ephemeral) or ECDHE (elliptic curve Diffie-Hellman ephemeral) tags.

This functionality prevents key disclosure laws and server seizure from resulting in the decryption of previously recorded traffic. TLS configurations supporting forward secrecy increased in popularity after fears that signals intelligence agencies were recording and storing encrypted traffic for later analysis, through the BULLRUN and EDGEHILL programmes[31].

Perfect Forward Secrecy

Perfect forward secrecy is a more secure form of forward secrecy where ephemeral keys are generated per-message, rather than per-session. It is achieved by piggy-backing key exchanges into each message, a practice known as key material ratcheting, used in protocols including Off-the-Record Messaging (OTR).

Unfortunately, the TLS specification does not draw a distinction between forward secrecy and perfect forward secrecy, so the terms are widely used interchangeably to refer to the non-perfect flavour. Data from the Trustworthy Internet Movement showed that as of January 2014, only 5% of the the top 200,000 most trafficked HTTPS-enabled web sites used forward secrecy by default[8].

2.3.2 Null Algorithms

The SSL/TLS protocol specifications offer cipher suites with null ciphers, and a test cipher suite[1] that offers no security whatsoever. Cipher suites with null ciphers offer no defence against eavesdropping, but still prevent message tampering and allow secure key exchange. These cipher suites should never be enabled for modern practical use of SSL/TLS.

2.3.3 Elliptic Curve Cipher Suites

In RFC 4492, elliptic curve cipher suites were added to all versions of TLS, introducing key exchange algorithms and client authentication based on elliptic curve cryptography (ECC)[9]. The major benefit of using ECC is that the security offered per bit is significantly higher than traditional public-key cryptography, so the relative computational cost is much lower. Table 2.1 details the estimated key sizes for comparable levels of security.

Groups of elliptic curve parameters, often referred to as "named curves," are standardised by several bodies including the National Institute of Science and Technology (NIST) and Brainpool

[1] `TLS_NULL_WITH_NULL_NULL`

Symmetric	ECC	DH/DSA/RSA
80	163	1024
112	233	2048
128	283	3072
192	409	7680
256	571	15360

Table 2.1: Comparable key sizes (bits) from RFC 4492[9]

(a conglomeration of German-speaking academics and companies). Like cipher suites, the responsibility of maintaining the list of named curves supported by TLS falls to IANA. The majority of the named curves supported at the time of publication of RFC 4492 were standardised by NIST, so the majority of TLS communication that uses ECC relies upon a few NIST curves.

2.4 Threats

2.4.1 Authentication Issues

Public Key Infrastructure

Authentication is seen as one of the biggest problems with TLS, due to the reliance on public key infrastructure and certificate authorities. This issue is both due to technical and organisational failures.

In HTTPS, client implementations rely on checking that the X.509 subject common name is the expected domain name. There is no technical barrier to prevent a certificate authority from validating a certificate for a domain name without the owner's permission. Once such a certificate has been created, an adversary can masquerade as the domain and clients will trust the adversary implicitly.

Additionally, the number of entities operating as certificate authorities entrusted to sign certificates faithfully and without error is large, and growing. The Electronic Frontier Foundation found that there are approximately 650 independent CAs with the ability to sign certificates that would be accepted by modern web browsers[10]. The number of CAs and the ease with which an adversary could purchase one makes it highly impractical to ensure that forged certificates cannot be be created.

Man-in-the-Middle Attack

After obtaining a forged certificate for a given domain, an active attacker can execute a man-in-the-middle attack and intercept plaintext traffic. Figure 2.5 shows how this is achieved by the composition of two SSL/TLS connections. Two sessions are maintained: one between the attacker

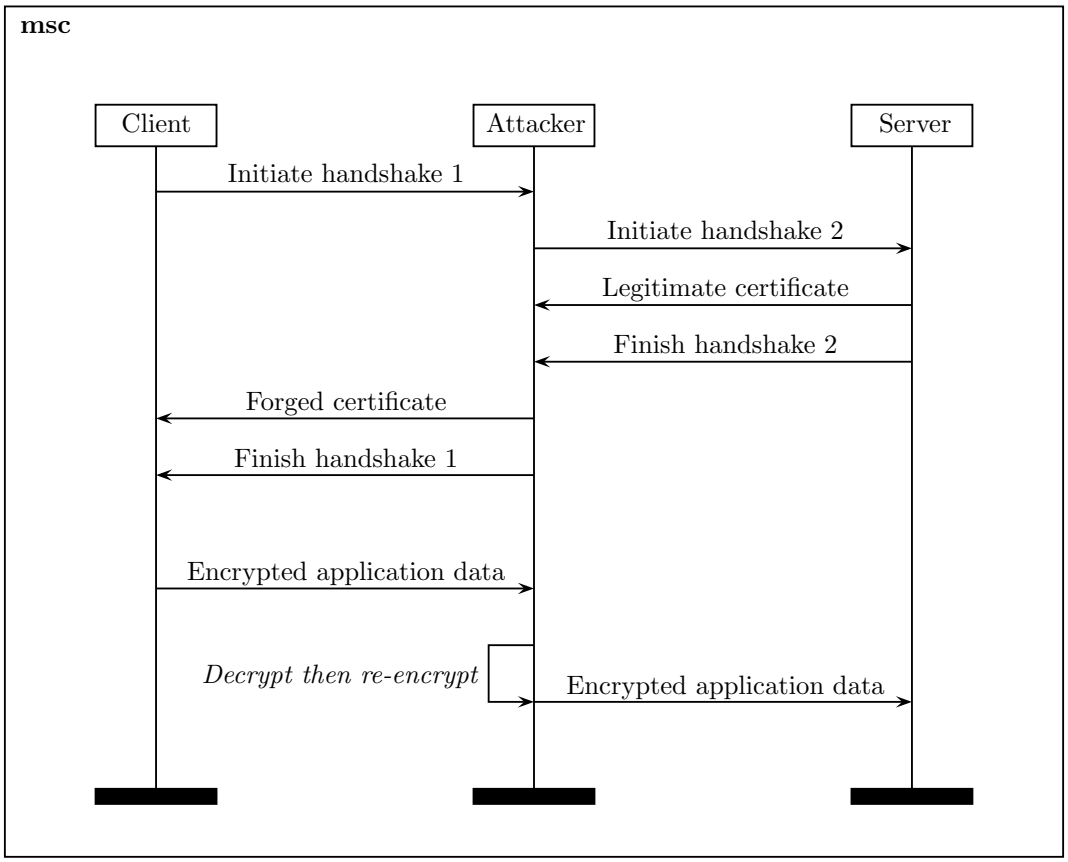


Figure 2.5: Man-in-the-middle attack

(masquerading as the server) and the client, and one between the attacker (masquerading as the client) and the server. Application data sent from the client is decrypted and can be read by the attacker before being re-encrypted and sent to the server.

In 2011, the Islamic Republic of Iran performed this attack to decrypt traffic between Google's Gmail and its citizens[11], evidencing the viability of this practical attack.

Certificate Pinning

Certificate pinning is a defence against man-in-the-middle attacks. It works by modifying clients to verify that public-key fingerprints in X.509 certificates match known valid values, rather than checking the subject CN field. With this approach, authentication relies upon the collision-resistance of the cryptographic hash functions used to generate public key fingerprints, making the role of a certificate authority redundant.

However, the problem with this defence is using a secure mechanism for communicating the set

of valid fingerprint values. Google has enabled certificate pinning in the Chrome web browser for its HTTPS-enabled services, and to solve this problem bundles valid fingerprint values with their installation binaries[12].

2.4.2 Cryptographic Vulnerabilities

CBC Padding

In 2002, Serge Vaudenay published a paper documenting security flaws in protocols (including SSL/TLS) that utilise padding and a block cipher in CBC mode[16]. The documented attack was shown to induce a side channel (see Section 2.4.4) for TLS v1.0 and below.

The attack relies on the fact that when the client receives a message its payload is decrypted before validating the MAC. If the padding of the payload is invalid, a protocol alert message is sent to the sender, otherwise an acknowledgement is sent. So if an adversary can differentiate the two responses, they can gain knowledge about the plaintext.

A fix was proposed that appended a block containing $h(m||p)$ to the plaintext, where h is a hash function, m is the message, and p is the padding required to align to a block multiple. This meant that clients would decrypt then verify the hash before checking the padding. Consequently, the TLS v1.0 specification was revised to TLS v1.1 to fix this vulnerability[5] and it was proposed that older versions consider disabling cipher suites operating under CBC mode.

Stream Cipher Keystream Reuse

Stream ciphers that utilise exclusive-or and reuse keys are vulnerable to an attack that may permit the decryption of message contents. Using the commutativity of exclusive-or, an attacker can deduce A $\oplus$ B of two messages A and B that were encrypted under the same keystream values. If either message is known, the other can be decrypted, and if neither are known frequency analysis may be used to decrypt message contents.

This attack can be mitigated by using per-message IVs to derive different keystreams from a static session key. The size of the IV must also be considered in relation to the number of messages exchanged before re-keying; the 24-bit IV specified in the Wired Equivalent Privacy (WEP) protocol was too small and thus overflow caused keystream reuse, breaking the security of the protocol.

SSL has used per-message IVs since v3 and random per-message IVs since TLS v1.1, so this attack is no longer viable. However, a similar attack for SSL v3 and TLS v1.0 was based on the predictability of IVs (Section 2.4.3).

Stream Cipher Malleability

Malleability is a property of a cryptosystem that allows an attacker to construct a ciphertext, from observed ciphertexts, that decrypts to a valid plaintext. Figure 2.6 demonstrates a 'bit-flipping'

attack on messages encrypted under a stream cipher that utilises exclusive-or and thus has this property.

If an attacker can correctly guess a subsection of message, they just need to exclusive-or the ciphertext with (known subsection) $\oplus$ (new subsection) to get a another valid ciphertext. A poorly implemented client may naïvely assume that because the key is only known by itself and the server, that the server can be the only other entity to generate valid ciphertexts.

This attack can be mitigated by using MACs that use a strong hash function. Message authentication is present from SSL v2 onwards, with improvements in SSL v3 and TLS, so this attack is no longer viable.

RC4 Keystream Bias

In 2013, AlFardan et al. published a paper titled *On the Security of RC4 in TLS and WPA*. This detailed a single-byte statistical bias in the first 256 bytes of RC4's keystream, which coupled with enough ciphertexts of a fixed plaintext encrypted under different keys, can lead to the recovery of up to 220 bytes of TLS plaintext[17]. (An ideal stream cipher should produce a keystream that is indistinguishable from random noise, to imitate the information-theoretically secure one-time pad.)

The paper states that *"In our experiments, the first 40 bytes of TLS application data after the* `Finished` *message were recovered with a success rate of over 50% per byte, using* 2^{26} *[TLS] sessions. With* 2^{32} *sessions, the per-byte success rate is more than 96% for the first 220 bytes (and is 100% for all but 12 of these bytes)."* For HTTPS, it was shown that—given enough ciphertexts—session cookie data could be efficiently recovered from a session encrypted with RC4[17].

Therefore cipher suites containing RC4 are no longer considered to be secure against practical attacks. This poses a serious problem, as RC4 has achieved significant adoption; its simplicity means that it is computationally efficient so is favoured by system administrators, it has been supported in all versions of SSL/TLS to date, and authors of previous vulnerabilities have recommended using RC4 as a workaround to fix other issues.

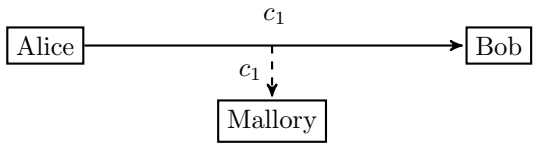
NIST/Brainpool Named Elliptic Curves

As was mentioned in Section 2.3.3, most TLS implementations that support elliptic curve cryptography support the NIST—and to a lesser extent—Brainpool named elliptic curves. There exists significant scepticism regarding the methodology used to generate the parameters of these standardised curves, especially given the 2014 publication of details regarding the US National Security Agency's attempts to weaken cryptography through its BULLRUN programme[31].

Let Figure 2.7 define a methodology to generate standardised elliptic curves.

1. Alice encrypts message m_1, "`SEND$100`", under a stream cipher that generates the pseudo-random keystream $S(k)$. Ciphertext c is constructed bitwise, by combining the ith bit of the message with the ith bit of the keystream using the exclusive-or operator: $c = m \oplus S(k)$. Alice encrypts m_1 and sends the corresponding ciphertext c_1 to Bob.

	'S'	'E'	'N'	'D'	'$'	'1'	'0'	'0'
m_1	01010011	01000101	01001110	01000100	00100100	00110001	00110000	00110000
$S(k)$	00001111	10000101	10100111	10100100	00001010	11001001	10111111	01000100
c_1	01011100	11000000	11101001	11100000	00101110	11111000	10001111	01110100



2. Mallory records c_1 and, without knowing the entire contents of the plaintext m_1 nor keystream S, is able to construct a modified ciphertext c_2 that corresponds to the plaintext "`SEND$999`". Mallory sends c_2 to Bob.

						'9'	'9'	'9'
m_2						00111001	00111001	00111001
m_1						00110001	00110000	00110000
$m_2 \oplus m_1$						00001000	00001001	00001001
c_1	01011100	11000000	11101001	11100000	00101110	11111000	10001111	01110100
c_2	01011100	11000000	11101001	11100000	00101110	11110000	10000110	01111101

3. Bob receives c_2, and decrypts it using the generated keystream S, successfully yielding valid plaintext m_2.

c_2	01011100	11000000	11101001	11100000	00101110	11110000	10000110	01111101
$S(k)$	00001111	10000101	10100111	10100100	00001010	11001001	10111111	01000100
m_2	01010011	01000101	01001110	01000100	00100100	00111001	00111001	00111001
	'S'	'E'	'N'	'D'	'$'	'9'	'9'	'9'

Figure 2.6: Anatomy of a stream cipher 'bit-flipping' attack

1. Take a prime p and elliptic curve parameterised by constant B, say, $x^3 - 3x + B \mod p$.
2. Let $B = H(s)$, where s is our 'seed' and H is a cryptographic hash function, say SHA-1.
3. Assign a random value to s and hence calculate B.
4. Assess the security of the elliptic curve against all known attacks. If it is known to be weak, go back to step 3, else stop.
5. For various primes, publish p, H, s, $x^3 - 3x + b \mod p$ as a named curve standard.

Figure 2.7: A verifiably random methodology for generating standardised elliptic curves

The justification is that, assuming that H cannot be inverted, introducing a hidden vulnerability into the elliptic curve by careful choice of s is so computationally intensive that it was considered to be impossible. This approach is termed "verifiable randomness" in various standards and is the methodology NIST used to generate their suite of named curves. However, it has attracted criticism because there is no explanation of the choice of seeds and no way to verify that they were chosen at random.

Critics[32] claim that an ECC and/or SHA-1 vulnerability known only the authors of the standard and access to a enough brute force computing power could mean that s values have been chosen such that the curves have been intentionally weakened. To mitigate this, the Brainpool standard uses "nothing up our sleeves" numbers, where seeds are deterministically generated.

Let Figure 2.8 define an alternative, more secure, methodology to generate standardised elliptic curves.

1. Take a prime p and elliptic curve parameterised by constants A and B, say, $y^2 = x^3 - Ax + B \mod p$.
2. Let $A = H(s)$, $B = \overline{A}$, where s is our 'seed' and H is a cryptographic hash function, say SHA-3 (512-bit).
3. Let s be composed of a counter c, say of length 32-bits, followed by the digits of a transcendental number, say cos(1), taken as an integer and truncated
4. Let $c = 0$
5. Assess the security of the elliptic curve against all known attacks. If it is known to be weak, increment c and repeat, else stop and publish the parameters as a standard.

Figure 2.8: A verifiably pseudorandom methodology for generating standardised elliptic curves

The idea is that cos(1)—or any transcendental number, for that matter—has no causal relationship with the security of the resulting elliptic curve and now the adversary's hidden vulnerability must coincide with the first value of c that is secure against all known attacks, a *much* smaller search space to work with. Being able to systematically and comprehensively explain seed values achieves "verifiable pseudorandomness" and the Brainpool named curves specification fulfils this, using a similar methodology to above and transcendental numbers π and e.

Critics are still sceptical of the Brainpool named curves given that new attacks against elliptic curve cryptography have been discovered since their standardisation and the specification uses the ageing (NIST-standardised) SHA-1, though these are not the largest concerns with Brainpool's approach.

In 2014, Bernstein et al. showed that "verifiable pseudorandomness" is not strict enough to prevent an adversary from engineering a vulnerability into the resulting elliptic curve by producing a suite of standard curves with the value `0xBADA55` engineered into their constants. Given the exact methodology as defined in Figure 2.8 with $p = 2^{224} - 2^{96} + 1$ (as in the NIST P-224 curve), we get $c = 184$ meaning that:

```
s = 000000B8 8A51407D A8345C91 C2466D97 6871BD2A

A = 7144BA12 CE8A0C3B EFA053ED BADA555A
    42391FC6 4F052376 E041C7D4 AF23195E
    BD8D8362 5321D452 E8A0C3BB 0A048A26
    115704E4 5DCEB346 A9F4BD97 41D14D49
```

This is analogous to showing that the resulting elliptic curve could be engineered to contain a vulnerability known only to the creators. As a result, we cannot trust verifiably pseudorandom named elliptic curves and therefore have exhausted the named curves specified in the ECC extension to TLS. RFC 4492 does specify the option to use arbitrary explicit elliptic curves, but this does not solve our problem of identifying a 'secure' elliptic curve. Out of a general sense of conservatism, I am personally concerned about the increased usage of elliptic curve cryptography to secure TLS connections.

2.4.3 SSL/TLS-Specific Vulnerabilities

Client-Initiated Renegotiation

In 2009, Marsh Ray published details of a man-in-the-middle attack that allows an adversary to prefix application-level plaintext to the requests of a client[18]. It exploits the client-initiated renegotiation feature of the SSL/TLS protocol and relies on the buffering of pending application-level requests on the server.

TLS session renegotiation allows a client to perform standard handshakes to renew cryptographic session parameters, over an existing encrypted session. A man-in-the-middle can exploit this as follows:

1. The attacker establishes a connection with the server, and exchanges partial application-level data that is intended to be prefixed to eventual requests made by the client. (For example, this could modify an HTTPS GET to request a page of the attacker's choice.)
2. The attacker waits until the client attempts to initiate a connection with the server. It intercepts the handshake, and requests session re-negotiation for its existing connection with the server.
3. The attacker then forwards messages such that the client's handshake messages are used in the session re-negotiation.
4. The client makes an application-level request that the server believes to be a continuation of the partial request the attacker made previously.
5. The server returns the result of the attacker's prefixed request to the client.

This vulnerability does not allow the attacker to view the response (nor modify it), only to prefix data to *r*equests. This attack is possible because no binding exists between the re-negotiation request and the enclosing encrypted channel. An internet draft suggests protocol modifications to fix this issue[19] and the workaround suggested until these changes are adopted is to fully disable client-initiated renegotiation.

Browser Exploit Against SSL/TLS (BEAST)

BEAST is a proof of concept attack based on the weaknesses discovered in SSL v3 and TLS v1.0 that result from the predictable use of implicit IVs. Though the concept was introduced in 2006 by Gregory Bard in *A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL*[20], it was not until 2011 that this was turned into a practical attack on TLS by Rizzo and Duong[21].

BEAST is a client-side attack where session cookies can be recovered from the ciphertext of an HTTPS connection that has negotiated a block cipher in CBC mode. It requires the attacker to have the ability to inject known plaintext into the requests made by the client to the server, achieved using cross-origin HTTP requests in the BEAST proof-of-concept code.

If it is known that some value is located in a block m_i, then an attacker will guess that this is equal to value v. The attacker then chooses to inject the next plaintext block to be encrypted: $x \oplus c_{i-1} \oplus v$ (where x is the next IV to be used and c_{i-1} is the previous block of ciphertext). Before encryption, this block has the IV applied, so that the encryption process actually takes place on $c_{i-1} \oplus v$, to produce c_{i+1}. If the guess for v was correct then it can be observed that $c_i = c_{i-1}$.

Rizzo and Duong reduce the search space of this value v to that of one byte, by injecting other data such that v is split into two blocks, where the first block contains only known data plus the first byte of v. They perform an exhaustive search to find this first byte, then adjust the offset accordingly and repeat, until the entire plaintext value of v has been recovered.

The vulnerability that permits this was fixed in TLS v1.1 by introducing random per-message IVs[5]. As a workaround for TLS v1.0, forcing the use of RC4 mitigates this attack since it is a stream cipher and therefore not susceptible. However, as mentioned in Section 2.4.2, RC4 is now considered insecure, so there is currently no possible safe configuration for SSL/TLS versions older than TLS v1.1. That said, this attack is not feasible unless an adversary can inject arbitrary plaintext to be encrypted.

Compression Ratio Info-leak Made Easy (CRIME)

CRIME is another attack discovered by the authors of BEAST and presented at the Ekoparty security conference in 2012[22]. It relies on TLS compression (DEFLATE) being enabled, the ability to inject arbitrary plaintext data and can be used to efficiently recover session cookies for HTTPS. It works as follows:

1. The attacker makes an HTTPS request from the client to some URL such that it is known that the cookie named c will be sent in the request. The cookie name and a guessed first value is appended to the request URL as a parameter, e.g., "`GET /?c=a`."
2. The attacker notes the length of the encrypted packet sent.
3. The attacker makes subsequent HTTPS requests to the same URL with the appended variable changed, e.g., "`GET /?c=s`". If the size of the request does not increase, the value of the variable is likely the prefix of the cookie.
4. The attacker repeats this process by appending to the variable until the value of the cookie is known. e.g., "`GET /?c=secret`", which refers to the encrypted HTTP header "Cookie: c=secret."

The workaround suggested is to fully disable TLS compression. However, this attack inspired a subsequent attack, BREACH, which simply relies on HTTP compression enabled at the application-level to achieve the same goal.

Lucky Thirteen

Published in 2013, Lucky Thirteen affected common implementations of all TLS versions, but was limited to connections secured by block ciphers using CBC mode and the HMAC-SHA1 algorithm[23].

This attack was built upon the work of Vaudenay (Section 2.4.2) who discovered that knowing whether padding was formatted correctly (through what is referred to as a padding 'oracle') allows an attacker to infer plaintext values via statistical analysis. Lucky Thirteen shows that a man-in-the-middle can induce a padding oracle by truncating ciphertexts and altering two bytes per packet[23].

This is because Vaudenay's previous side channel discovery caused implementers to force MAC checking even for ciphertexts with known incorrect padding. By truncating packets, the authors of Lucky Thirteen are able to distinguish timing differences between two types of messages: ones where the end of the truncated plaintext *looks like* padding, and other messages that look like they do not. If the client mistakenly removes what it thinks is padding, then the HMAC-SHA1 algorithm takes fewer cycles to execute and an alert message is sent earlier than expected, inducing a side channel for an adversary to exploit.

The authors claim that trying all combinations of two-byte alterations guarantees them the ability to recover two bytes of plaintext. While this attack is impressive, it is unlikely that it has practical application over a network environment such as the internet. In addition, every two-byte alteration sent causes the client to disconnect and re-establish an SSL/TLS session, which is noticeable and very time-consuming.

Furthermore, popular SSL/TLS implementations have been patched to prevent this timing attack, and it is ineffective if a stream cipher or AEAD is used.

2.4.4 Side Channel Attacks

A side channel attack is generally defined to be an attack based upon information gained through the weaknesses in the implementation of a cryptosystem as opposed to algorithmic vulnerabilities or cryptanalysis. Side channels include execution timing, power usage and data remanence (e.g., cache contents).

Entropy Issues

Most cryptography relies on the availability of a source of unpredictable random numbers (with high entropy) for security against attacks. Systems are unable to participate in secure communication if too little entropy is available to generate secure random numbers, thus there are a number of approaches taken to 'feed' the entropy pools of a system. These include:

- Hardware security modules (HSMs) or trusted platform modules (TPMs). These are able to generate true random numbers through a hardware-based process.
- Hard drive seek timings[24]. In 1994, Davis et al. demonstrated that the air turbulence inside a hard disk could be used to extract 100 independent random bits per minute.
- Mouse and keyboard interactions; measuring the time deltas between keystrokes.

- Uninitialised RAM. This approach was used as a seed for OpenSSL's pseudo-random number generator (PRNG).

In 2006, an update was made to Debian's OpenSSL package that removed references to uninitialised memory addresses[25]. This unintentially removed part of the seeding process for the PRNG, massively reducing the randomness of data used to generate cryptographic keypairs. The only data used to seed the PRNG was the process ID, which has a maximum value of 32,767, and this severely reduced the set of possible keypairs to such an extent that all possible weak keypairs have been hashed and enumerated in the openssl-blacklist package. Potentially, certificates sent as part of an SSL/TLS handshake could be checked against this blacklist.

In 1996, Netscape's web browser utilised a similarly-insecure proprietary algorithm to seed their PRNG for SSL connections. Two security researchers, Ian Goldberg and David Wagner, reverse-engineered the browser binary to discover that, by design, the RNG was only seeded with the time of day and parent/child process IDs[26]. These three values are predictable and not independent, so had very low entropy, which undercut the security model of SSL.

Web Traffic Leaks

In *Preventing Side-Channel Leaks in Web Traffic: A Formal Approach* Backes et al. create a mathematical model to show that the structures of websites can permit the fingerprinting of their usage despite data being encrypted over SSL/TLS[27]. Side channels including request ordering, packet sizes and delays all give away information that can be used classify the behaviour of users.

The attacks are practical and were demonstrated to predict values typed into an interactive autocomplete field, as well as the pages visited on the Bavarian-language version of Wikipedia. They further demonstrate a *generalised* model for quantifying the information leaked through side channels and "techniques for the computation of security guarantees" for all possible execution paths of a web application[27].

They claim that their model is able to test mitigations of attacks launched by realistic adversaries, as opposed to alternatives that draw conclusions about countermeasures from the viewpoint of single-purpose classification algorithms[27].

The authors mention a number of useful possible countermeasures that can be applied below or at the TLS layer, including:

- Padding: up to 255 bytes of padding can be added per TLS message.
- Dummy data: TCP retransmission could help mitigate aggregate size-based attacks.
- Split: TCP packets could be segmented to avoid size-based side channel attacks.
- Shuffle: requests could be buffered and their order changed before being sent.

2.4.5 Notable Implementation Bugs

OpenSSL Heartbleed (CVE-2014-0160)

In April 2014, a memory bounds checking bug was identified in the TLS heartbeat functionality of the popular library OpenSSL. This bug allows attackers to retrieve up to 64KB of server memory and it has been shown that this ability permits the practical theft of session data and private keys related to X.509 certificates.

Data from Netcraft[2] suggests that 500,000 TLS-enabled web servers were vulnerable to Heartbleed, which resulted in system administrators scrambling to apply patch sets and reissue certificates before being exploited. Worryingly, further data from Errata Security[3] showed that 300,000 systems were still vulnerable to Heartbleed one month after the bug's initial publication.

TLS heartbeat messages contain a variable length payload and a 16-bit integer explicitly specifying the payload length. In response to a heartbeat request message where the payload is smaller than the length specified, a vulnerable OpenSSL instance performs a buffer over-read and returns the data that happens to exist in memory beyond the payload. The solution to this bug is very simple: verify that heartbeat messages do not contain explicit lengths that exceed the size of the payload or, given the marginal use of TLS heartbeat functionality, disable heartbeat messages altogether.

See Section 3.3.1 for details of TLSFilter's mitigation technique.

OpenSSL ChangeCipherSpec Injection (CVE-2014-0224)

In June 2014, another serious implementation bug was identified in OpenSSL as a direct result of the increased scrutiny from the security community in the wake of Heartbleed. If both a client and server were running a vulnerable version and a man-in-the-middle were to inject a TLS ChangeCipherSpec message into a handshake prior to the shared master secret being populated with data, the handshake would terminate early and the session would be encrypted under an empty key.

While this is a severe vulnerability, it relies on having a privileged network position between the client and server meaning that, unlike Heartbleed, it cannot be exploited by everyone. The solution to this bug is to verify that the TLS handshake state is such that a ChangeCipherSpec message should be processed if received or otherwise ignored.

A TLSFilter plugin was written to mitigate this vulnerability prior to the public availability of tools to test for it.

[2] http://news.netcraft.com/archives/2014/04/08/half-a-million-widely-trusted-websites-vulnerable-to-heartbleed-bug.html

[3] http://blog.erratasec.com/2014/05/300k-servers-vulnerable-to-heartbleed.html

2.5 Similar Existing Works

Despite a thorough search of IEEExplore, the ACM digital library, Citeseer, ScienceDirect and Google Scholar, I was unable to identify any academic works regarding the use of firewalling/proxies/middleware to improve the *security* of SSL or TLS. A class of tangentially similar *products* exist and are known as intrusion detection systems (IDS), though these focus on the monitoring and alert of attack attempts as opposed to prevention.

2.5.1 Bro

According to the Bro Project website[4], *"Bro provides a comprehensive platform for network traffic analysis, with a particular focus on semantic security monitoring at scale. While often compared to classic intrusion detection/prevention systems, Bro takes a quite different approach by providing users with a flexible framework that facilitates customized, in-depth monitoring far beyond the capabilities of traditional systems. With initial versions in operational deployment during the mid '90s already, Bro finds itself grounded in more than 15 years of research."*

Bro is an entirely passive piece of software and therefore not well-suited to firewall applications in which packets require active manipulation. Furthermore, it will opt to drop network packets from its processing pipeline in preference to impacting network throughput when CPU consumption and memory limits are reached, which is a restriction that wholly prevents its use as a firewall.

With regards to extensibility, Bro can be fully customised through its support of a proprietary scripting language. In addition, a large corpus of user-authored scripts provide a large breadth and depth of functionality at no cost.

2.5.2 Libnids

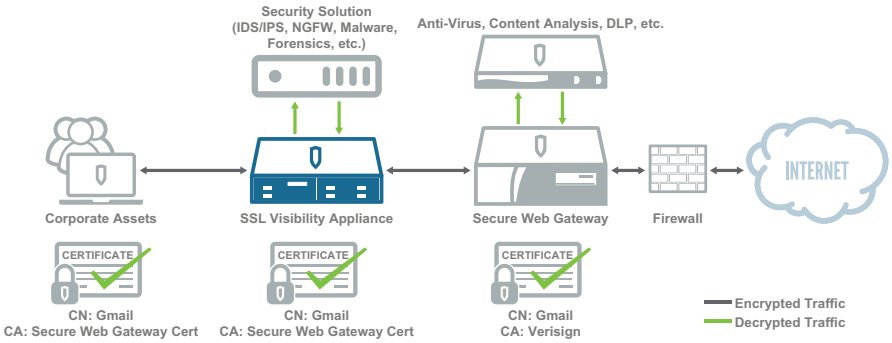
According the the Libnids website, "*Libnids is an implementation of an E-component of [a] Network Intrusion Detection System. It emulates the IP stack of Linux 2.0.x. Libnids offers IP defragmentation, TCP stream assembly and TCP port scan detection.*" It is specifically intended to be used as the basis of network intrusion detection systems, so would also require substantial modification before being used as a firewall.

Unlike Bro, Libnids focuses on a very narrow set of features; while there is a basic C API for registering callback functions, it lacks the flexibility and ease of extension that Bro supports. Additionally, Libnids has been abandoned since 2010 and is in need of bug fixes before being functional with the current Linux kernel. Since this project is geared towards being an open source library and not an entire appliance aimed at end users, it is less easy to compare to TLSFilter.

[4] http://www.bro.org/documentation/faq.html

2.5.3 Blue Coat SSL Visibility

Blue Coat Systems is a commercial provider of enterprise security appliances. Their "SSL Visibility" product[5] allows enterprises and law enforcement the ability to monitor the contents of encrypted TLS traffic on their networks. It does this through a hardware network appliance that performs an intentional man-in-the-middle attack on all network users, with a new certificate for each website dynamically generated and signed under their "Secure Web Gateway Cert" certificate authority.



Unlike TLSFilter, the purpose of this product is not to improve security but to enable surveillance for use cases including corporate policy enforcement and network forensics. TLSFilter will attempt to maintain the end-to-end encrypted nature of all TLS connections, so as to not to break the security assumptions of its users. A drawback of Blue Coat's approach is that each device must have their custom CA certificate installed to the operating system's trusted root certificate store.

[5] https://www.bluecoat.com/products/ssl-visibility-appliance; Diagram courtesy Blue Coat

Chapter 3

TLSFilter Feature Set

This chapter describes the features implemented in TLSFilter with example rule configurations and analyses how they can be used to mitigate many of the TLS weaknesses previously described. Section 3.3.3 contains a table summarising the software's effectiveness against known TLS vulnerabilities.

3.1 TLS Record Layer Features

The TLS record layer is the protocol that sits on top of TCP (or UDP, in the case of Datagram TLS) and segments TLS data into variable-length versioned-and-typed records. This layer also fragments and re-assembles data, much like the the lower level TCP, as well as maintaining the guarantee of in-order records.

As of May 2014, the valid record types and their purposes are as follows:

- **Handshake**: indicates the record contains TLS handshake protocol messages and should be further processed.
- **Alert**: indicates a (possibly fatal) problem with the TLS connection, details of which are defined by the TLS alert protocol.
- **Application**: indicates that the record contains application-level data that the TLS implementation should skip processing. For example, for HTTPS these records would contain the HTTP requests and responses, encrypted under the parameters negotiated by the handshake protocol.
- **ChangeCipherSpec**: indicates that all following messages are to be encrypted and acts as a precursor to the end of the TLS handshake process.

- **Heartbeat**: built as part of a TLS extension to ensure that long-term TCP connections are maintained.

For the purposes of TLSFilter, alert records are excluded from processing as the underlying TLS endpoints will reattempt or close connections after having received a record of this type. Similarly, application records are excluded from the processing pipeline since the contents is encrypted and highly-specific to the protocol wrapped by TLS.

While the core functionality allows for the blacklisting/whitelisting of TLS versions, to perform this at the record layer is inappropriate. TLS actually performs version negotiation as part of the handshake protocol, so we cannot assume that the protocol version in the record layer will be the version used to secure communication.

Most—but not all—processing is performed on handshake records, so the bulk of the TLSFilter implementation uses an architecture that abstracts away the record layer and operates purely on the higher-level handshake protocol. In contrast, an example of lower-level logic at the TLS record layer is exemplified in the heartbeat plugin in Section 3.3.1.

3.2 TLS Handshake Protocol Features

The TLS handshake protocol (documented in Section 2.1.4) allows the secure negotiation of TLS parameters used to set up the channel before application data can be exchanged. Through the `parameter` configuration keyword, TLSFilter allows users to allow and deny connections on the basis of the parameters negotiated throughout the handshake process.

In general, the client—that is, the endpoint that initiates the connection—will inform the server of its capabilities through a ClientHello handshake message and allow the server final say in the parameters used for communication, which are communicated back through a ServerHello handshake message. To maintain maximum interoperability, TLSFilter takes a permissive approach and will only issue a verdict on the basis of final communication parameters.

For example, an insecure client could initiate a TLS connection with NULL and/or insecure cipher suites as its top preferences and this is permitted unless the server decides to *select* one of these insecure preferences. The drawback of this approach is that connections where *all* client capabilities are insecure are bound to fail, but suffer increased latency between the connection being solicited and denied as this requires, at minimum, an extra round-trip from the server.

The principle of delaying a verdict until as late as possible has been repeated throughout the architecture of TLSFilter.

3.2.1 Cipher Suite-Based Rules

Cipher suites determine the fixed sets of algorithms used for encryption, message authentication and key exchange. Hence, TLS weaknesses caused by vulnerabilities in cryptographic algorithms

(or classes thereof) can be prevented by denying groups of cipher suites. TLSFilter supports the denial of cipher suites on the basis of: suite name, symmetric cipher, key exchange algorithm, MAC digest algorithm and cryptographic modes of operation, as well as supporting macros for "export grade" suites and those supporting forward secrecy.

Suite Name

Cipher suites are allowed or denied on the basis of IANA-assigned names. For example, the following configuration file excerpt prevents a TLS handshake from negotiating the NULL cipher suite (a suite that should never be used in production):

```
parameter deny suite TLS_NULL_WITH_NULL_NULL
```

Symmetric Cipher

One may also define rule sets on the basis of the symmetric ciphers used. For example, to mitigate the potential for keystream biases (Section 2.4.2) in the RC4 stream cipher from compromising security, the following rule could be used:

```
parameter deny cipher RC4
```

which is equivalent to

```
parameter deny suite TLS_RSA_EXPORT_WITH_RC4_40_MD5
parameter deny suite TLS_RSA_WITH_RC4_128_MD5
parameter deny suite TLS_RSA_WITH_RC4_128_SHA
parameter deny suite TLS_DH_anon_EXPORT_WITH_RC4_40_MD5
parameter deny suite TLS_DH_anon_WITH_RC4_128_MD5
parameter deny suite TLS_KRB5_WITH_RC4_128_SHA
parameter deny suite TLS_KRB5_WITH_RC4_128_MD5
parameter deny suite TLS_KRB5_EXPORT_WITH_RC4_40_SHA
parameter deny suite TLS_KRB5_EXPORT_WITH_RC4_40_MD5
parameter deny suite TLS_PSK_WITH_RC4_128_SHA
parameter deny suite TLS_DHE_PSK_WITH_RC4_128_SHA
parameter deny suite TLS_RSA_PSK_WITH_RC4_128_SHA
parameter deny suite TLS_ECDH_ECDSA_WITH_RC4_128_SHA
parameter deny suite TLS_ECDHE_ECDSA_WITH_RC4_128_SHA
parameter deny suite TLS_ECDH_RSA_WITH_RC4_128_SHA
parameter deny suite TLS_ECDHE_RSA_WITH_RC4_128_SHA
parameter deny suite TLS_ECDH_anon_WITH_RC4_128_SHA
parameter deny suite TLS_ECDHE_PSK_WITH_RC4_128_SHA
```

Key Exchange & Authentication Algorithm

The key exchange and authentication algorithm is the functionality defined by the cipher suite to ensure that the session keys to be used to encrypt the connection under a symmetric cipher are exchanged securely. The algorithm selected by the server in the ServerHello handshake message determines the existence and formats of subsequent ClientKeyExchange and ServerKeyExchange handshake messages.

For example, to prevent a trivial man-in-the-middle attack, the unauthenticated Diffie-Hellman key exchange algorithm can be disabled using:

```
parameter deny algorithm TLS_DH_anon
```

Digest Algorithm

The cryptographic hash function defined in the cipher suite has multiple uses and depends on the TLS (or SSL) version negotiated. In TLS 1.2, the negotiated hash algorithm is used in the pseudorandom function to generate the shared session secret, in message signatures via the HMAC construct and for the Finished message payload that signals the end of the handshake protocol. (Of course, this assumes the use of a non-AEAD cipher suite where authentication and encryption are distinct processes.)

In versions earlier than 1.2, the pseudorandom function, *handshake* message signatures and the Finished message payload use both SHA1 and MD5 regardless of the cipher suite negotiated. The hash function specified in the cipher suite is relegated to the HMAC (or in the case of SSL 3.0, HMAC-like algorithm) that ensures the integrity and authenticity of *application* data.

However, unlike naïve signatures, the HMAC construct is not vulnerable to collision attacks. Even though MD5 is no longer considered cryptographically secure for signatures, RFC 6151 states *"The attacks on HMAC-MD5 do not seem to indicate a practical vulnerability when used as a message authentication code"*[33] but warns against continuing to support HMAC-MD5 in new protocols. Hence this feature is largely to be used to protect against future hash function vulnerabilities.

To follow the advice set out in RFC 6151 use:

```
parameter deny algorithm TLS_MD5
```

Mode of Operation

The cipher mode of operation is the last parameter specified in the cipher suite. This can be used to protect against the CBC padding attacks discovered by Vaudenay (Section 2.4.2) as follows:

```
parameter deny cipher-mode CBC
```

Macros

Two macro rules `export-grade` and `forward-secrecy` are defined to make it easier to deny (or require) cipher suites encompassed by these properties. The relaxation of cryptographic export laws—at least in the United States—means that using 'export grade' cipher suites is no longer necessary. Regardless, the advancement of computer processors has ensured that these cipher suites are insecure and thus should not be used under any technical use case.

```
parameter deny export-grade
```

The opposite is true of cipher suites that support forward secrecy through the use of ephemeral key exchanges. The advancement in technology and recent key disclosure laws such as RIPA now mean that it is practical for communications encrypted under TLS to be decrypted at a later date, en masse. Therefore, forward secrecy has become a desirable property of cryptosystems in general that seek to mitigate this attack vector.

```
parameter require forward-secrecy
```

3.2.2 Certificate-Based Rules

As we saw in Section 2.1.1, X.509 certificates are used to specify a public key and server identity binding which is signed by a trusted certificate authority. We saw there were numerous real-world weaknesses using this system for authentication, so TLSFilter offers several classes of rule based entirely on the X.509 certificate chains exchanged in a TLS handshake.

Public Key Modulus Size

A trivial vulnerability exists when a sufficiently large public key modulus is not used; a well-resourced adversary that commands enough processing power can factor RSA moduli and hence reconstruct private keys, violating the security model TLS relies upon. In 2009, Thorsten Kleinjung et al. successfully factored a 768-bit RSA modulus, although this took more than two years and used hundreds of computers[38].

Assuming the growth in processing power continues the minimum size of an RSA modulus considered 'secure' against the average adversary will continue to increase. Consequently, TLSFilter supports setting the minimum modulus size (in bits) for RSA, DSA and elliptic curve key pairs, for both server certificates and CA certificates. The following example requires RSA moduli to be a minimum of 4096 bits and ECC key pairs a minimum of 384 bits:

```
certificate key-length-minimum KP_RSA 4096
certificate key-length-minimum KP_ECC 384
```

Public Key Weakness

TLSFilter supports a blacklist of RSA public key hashes with all data currently derived from the openssl-blacklist database that was built in response to the Debian-OpenSSL PRNG debacle[25]. If future incidents weaken key pair generation and the keys can be enumerated, they can be trivally added to the blacklist in TLSFilter.

```
certificate deny-weak-rsa-keys
```

Pinning

As discussed in Section 2.4.1, certificate pinning can be used to mitigate man-in-the-middle attacks but until now has relied upon software developers checking at the application level. It does so by associating a hash of the entire X.509 certificate with the certificate common name. An ideal use case for this is functionality is software update servers:

```
# Ensure the Cyanogenmod Android ROM update server is not impersonated
certificate pin-sha1 download.cyanogenmod.org 5A:44:5B:2E:5B:D5:DA:98:3F:A1:FB:50:\
5C:6D:8A:53:49:02:D7:27
```

Without certificate pinning, an adversary could perform a man-in-the-middle attack with a certificate chain rooted by any certificate authority in the client's trust store. TLSFilter reduces the scope of this attack to that of a hash collision or modulus factorisation, rather than of third party coercion.

The realities of load-balancing often result in many valid certificate hashes for a single common name, so TLSFilter supports pinning in a many-to-one relationship:

```
certificate pin-sha1 encrypted.google.com ad:1f:f2:1a:4f:d7:9a:3b:58:b1:e2:ab:f9:6e:f5:ab
certificate pin-sha1 encrypted.google.com 16:04:f9:58:e3:49:bc:4c:c1:cc:cb:e7:f3:73:aa:da
certificate pin-sha1 encrypted.google.com 43:b4:cf:9c:d3:fc:c4:66:b5:a0:b2:7e:d7:e9:18:1d
```

Blacklisting

Similarly, certificates of servers and CAs can be blacklisted on the basis of their hashes. The following blacklists the DigiNotar CA certificates that were stolen and used to perform man-in-the-middle attacks on Google users in The Islamic Republic of Iran:

```
# Deny "DigiNotar Root CA"
certificate deny-sha1 C0:60:ED:44:CB:D8:81:BD:0E:F8:6C:0B:A2:87:DD:CF:81:67:47:8C
# Deny "DigiNotar Root CA G2"
certificate deny-sha1 43:D9:BC:B5:68:E0:39:D0:73:A7:4A:71:D8:51:1F:74:76:08:9C:C3
# Deny second "DigiNotar Root CA"
certificate deny-sha1 36:7D:4B:3B:4F:CB:BC:0B:76:7B:2E:C0:CD:B2:A3:6E:AB:71:A4:EB
```

```
# Deny "DigiNotar Services 1024 CA"
certificate deny-sha1 F8:A5:4E:03:AA:DC:56:92:B8:50:49:6A:4C:46:30:FF:EA:A2:9D:83
# Deny "DigiNotar PKIoverheid CA Overheid en Bedrijven"
certificate deny-sha1 40:AA:38:73:1B:D1:89:F9:CD:B5:B9:DC:35:E2:13:6F:38:77:7A:F4
# Deny "DigiNotar PKIoverheid CA Organisatie - G2"
certificate deny-sha1 5D:E8:3E:E8:2A:C5:09:0A:EA:9D:6A:C4:E7:A6:E2:13:F9:46:E1:79
```

3.2.3 Key Exchange-Based Rules

While the key exchange algorithm is defined by the cipher suite chosen by the server, the specifics of the key exchange are only known once the key exchange has been initiated. Specifically to ensure that ephemeral Diffie-Hellman key exchanges are of a sufficient level, the minimum modulus size can be set as follows:

```
keyexchange key-length-minimum dh 1024
```

It is important to note that a small modulus size entirely undercuts the benefit of using cipher suites that support forward secrecy. Forward secrecy mitigates a very specific type of attack, and without proper configuration of modulus minimums could lead to a situation where enforcing forward secrecy actually *weakens* security.

3.2.4 ECC Named Curves

TLSFilter also supports the blocking of specified named elliptic curves used during key exchange. Another nuance of forward secrecy is that the cipher suites that tend to be most popular use the Elliptic Curve Diffie-Hellman Ephemeral (ECDHE) key exchange algorithm, as ECC performance is an order of magnitude faster than non-EEC. Given that NIST named curves are the only curves widely supported by software implementations and that there is potential for these curves to contain intentional vulnerabilities (see Section 2.3.3), use of forward secrecy could again weaken security.

To block NIST's P-224 named curve, for example, use the following:

```
keyexchange deny named-elliptic-curve secp224k1
```

3.2.5 IP Subnet Profiles

Using the aforementioned rules, TLSFilter permits the creation of profiles that are applied on the basis of IP networks. Profiles are defined using a list of IP addresses and/or IP subnets in CIDR format, enclosed in square brackets:

```
[192.168.0.0/24, 127.0.0.1] # Rules below apply to 192.168.0.* and 127.0.0.1
...
```

In the source code and elsewhere in this document this may be referred to as a rule's 'scope'.

3.3 Application Programming Interface

Flexibility is a core property of TLSFilter, so to permit the case where a new TLS vulnerability cannot be mitigated using a combination of rules, custom functionality can be written the against the C plugin API. Plugins are dynamic objects that are loaded at runtime, so the extension of functionality does not require TLSFilter to be recompiled from source. All of the aforementioned rules have been implemented using the plugin interface, allowing the customisation of TLSFilter's functionality to any use case.

Plugins must fulfil the following requirements:

1. To have a function `void setup(ip_scope* s, uint8_t num_commands, char* command[])` which is automatically executed when TLSFilter loads the plugin. It is supplied with a list of strings equivalent to the configuration line pertaining to the plugin for a given—possibly `NULL`—scope.

2. Assign callback functions against TLS records, particular handshake messages or X.509 certificate chains by passing function pointers to:

   (a) `void register_record_callback(ip_scope* s, record_type rt, uint8_t (*fp)(tls_record*, tlsflow*))`

   (b) `void register_hs_callback(ip_scope* s, handshake_type ht, uint8_t (*fp)(tls_handshake_message*, tlsflow*))`

   (c) `void register_cc_callback(ip_scope* s, uint8_t (*fp)(X509*[], uint8_t))`

3.3.1 OpenSSL Heartbleed Mitigation

To evidence the extensibility inherent to TLSFilter's architecture, plugins were created to mitigate the OpenSSL Heartbleed (Section 2.4.5) and ChangeCipherSpec injection (Section 2.4.5) bugs once details of these vulnerabilities were made public. The source code of the heartbeat plugin fits into less than 50 lines of C:

```
#include "plugin.h"

#define DISABLE_TAG "DISABLE"
#define DENY_TAG "DENY"
#define HEARTBLEED_CLEARTEXT_TAG "HEARTBLEED-CLEARTEXT"

/* Wire format struct of a TLS handshake message (found within a TLS record) */
typedef struct __attribute__((__packed__)) tls_heartbeat_message {
    uint8_t type;      // 1 = request; 2 = response
    uint16_t length;   // Length of payload only
    void * payload;    // Followed by padding
```

```
} tls_heartbeat_message;

static uint8_t disable_heartbeat(tls_record* r, tlsflow* t){
    return NF_DROP;
}

static uint8_t deny_heartbleed_cleartext(tls_record* r, tlsflow* t){
    tls_heartbeat_message* hb = (tls_heartbeat_message*) &(r->payload);

    if (!t->encrypted && ntohs(r->length) < ntohs(hb->length) + 3 + 16){
        // We see a request where the TLS record length (after reassembly) is shorter
        // than the heartbeat payload length + header + padding
        return NF_DROP;
    }
    return NF_ACCEPT;
}

void setup(ip_scope* scope, uint8_t num_commands, char* command[]) {
    uint32_t i;
    for(i = 0; i < num_commands; i++){

        // Parse token(s)
        char* token[3];
        token[0] = strtok(command[i], " ");
        token[1] = strtok(NULL, " ");

        if(token[0] == NULL){
            fprintf(stderr, "Error: invalid heartbeat configuration");
            exit(1);
        }

        if (STRING_EQUALS(token[0], DISABLE_TAG)){
            register_record_callback(scope, HEARTBEAT, &disable_heartbeat);

        } else if (token[1] != NULL
                    && STRING_EQUALS(token[0], DENY_TAG)
                    && STRING_EQUALS(token[1], HEARTBLEED_CLEARTEXT_TAG)){
            register_record_callback(scope, HEARTBEAT, &deny_heartbleed_cleartext);
        }
    }
}
```

It is interesting to note that if a Heartbleed attack occurs after the TLS connection has transitioned to an encrypted state, it is not possible to ascertain whether the heartbeat request is legitimate. A heavy-handed solution to this is to deny *all* heartbeat records (since the TLS record layer header is never encrypted) at the expense of the few legitimate use cases where TLS heartbeat functionality is used.

Consequently, there are two TLSFilter rules to load this plugin with differing levels of heavy-handedness:

```
heartbeat deny heartbleed-cleartext
heartbeat disable
```

3.3.2 Configuration Context-Free Grammar

Appendix Figure A.1 details the context-free grammar that defines the syntax of TLSFilter configuration files, where `String` refers to an ASCII sequence that does not contain spaces and `EOL` refers to the ASCII line feed (LF) character.

3.3.3 Vulnerability Mitigation Table

Figure 3.1 provides a summary of TLSFilter's ability to mitigate known attacks against TLS.

Attack / Weakness	TLSFilter Rule	Caveat(s)
Man-in-the-Middle	`certificate pin-sha1`	Must know valid certificate hashes in advance
Known compromised CA	`certificate deny-sha1`	-
Vaudenay CBC padding	`parameter deny cipher-mode CBC`	Impacts reachability
RC4 keystream biases	`parameter deny cipher RC4`	-
Client-initiated renegotiation	None	No mitigation due to encrypted nature of channel
BEAST	`parameter deny cipher-mode CBC`	Impacts reachability
CRIME	`compression disable`	-
Lucky 13[a]	`parameter deny cipher-mode CBC`	Impacts reachability
Debian weak keys[b]	`certificate deny-weak-rsa-keys`	-
Private key disclosure[c]	`parameter require forward-secrecy`	Impacts reachability; should be used in conjunction with `keyexchange` rules
Weak cipher suites	`parameter deny export-grade`, `parameter deny algorithm TLS_NULL`	-
Weak private keys[d]	`certificate key-length-minimum` and `certificate ca-key-length-minimum`	-
Weak DH/DHE parameters	`keyexchange key-length-minimum dh`	-
Weak ECC named curve	`keyexchange deny named-elliptic-curve`	-
OpenSSL Heartbleed	`heartbeat disable` *or* `heartbeat deny heartbleed-cleartext`	Second rule cannot block encrypted attacks and first rule may impact genuine use of TLS heartbeat protocol

Figure 3.1: Summary of TLS weaknesses and appropriate TLSFilter rules

[a] Impractical attack
[b] RSA public keys
[c] An attack where encrypted traffic is recorded then later decrypted once private key material has been obtained
[d] RSA, DSA and ECC key pairs

Chapter 4

Design and Architecture

4.1 System Overview

Authoring TLSFilter required a breadth of technical knowledge in several areas: Linux networking, Linux kernel module development, low level IP, TCP and TLS protocol comprehension and enough of an understanding of the assumptions and weaknesses of the cryptography behind TLS to determine appropriate high-level rules. This chapter documents and justifies some of the higher-level architectural choices made with respect to these areas.

4.1.1 Operating System Integration

A firewall must integrate at a low-level with its host operating system, so the choice to support Linux was made out of a necessity to have well-documented, open access to the underlying OS functionality. Netfilter is the subsystem of the Linux kernel that processes network packets and provides hooks for kernel modules to interact with the network stack[28] and—along with the userspace tool `iptables`—has been the de facto Linux standard for firewalling since kernel version 2.3.

`nftables` is slated to replace, and generalise, the userspace functionality offered by `iptables` by providing a virtual machine interface upon which machine code can be run to perform packet inspection and modification. While this sounds like it would be well suited for TLSFilter, it is a very young project with limited support, so TLSFilter imitates the `iptables` model whereby a kernel module `tlsfilter.ko` hooks into the Netfilter subsystem and a userspace application `tlsfilter` is used for configuration and packet processing.

4.1.2 Fragmentation

Due to the realities of the Netfilter Linux kernel subsystem, TLSFilter receives *IP datagrams* and must issue a verdict on the basis of each of these packets. The low-level nature of this means that there are no guarantees with respect to packet ordering, defragmentation nor deduplication and thus TLSFilter is responsible for simulating the *entire network stack* between the relatively low-level IP layer and high-level TLS handshake protocol. Therefore great care was taken to prevent the transmission of any IP packet sequences that would allow a client or server to reconstruct an insecure TLS handshake protocol message or TLS record, or the entire security of the firewall would be undermined.

Rather than implementing a TCP/IP stack from scratch, I stripped-down and modernised a library (libnids) that uses TCP/IP reassembly code directly taken from version 2.0.36 of the Linux kernel to handle this process for me. I could then (somewhat) abstract my thinking to the TCP level, with a guaranteed stream of in-order, deduplicated and defragmented TCP segments, to which I would only need to account for fragmentation at the TLS record layer.

This approach guarantees that TLSFilter is as immune to all fragmentation-based evasion techniques as an, albeit dated, version of the Linux kernel. Ptacek and Newsham outline several intrusion detection evasion techniques in their 1998 paper[34] and show that the four most popular intrusion detection systems (IDS) are vulnerable to these techniques, stemming from the fact that TCP streams were reconstructed in a manner different to the destination systems.

It is important to note that, in theory, an additional layer of fragmentation can take place between the TLS handshake protocol layer and the TLS record layer. However, the SSL and TLS RFCs do not mention handshake protocol fragmentation so many implementations, including Microsoft's TLS support in Windows versions prior to 7, assume that a TLS handshake message will always be no bigger than a TLS record. The consequence of this is that most implementations do not fragment at this level, presumably to maintain compatibility. Since support was not universal and there is no practical use, I decided, like Microsoft, to omit this layer of handshake fragmentation; the consequence of is that the contents of handshake messages beyond 64KB is ignored.

4.1.3 Fragmentation Alternative

To avoid TCP low-level fragmentation concerns, the alternative approach is to intercept packets at a higher level in the Linux networking subsystem. Using the transparent proxying functionality (TPROXY) in kernel 2.2 onwards I could force the kernel handle the mechanics of TCP reassembly for me, drastically simplifying the scope of TLSFilter. However, there were several caveats to this approach:

- TPROXY usage is parameterised by port number, so realistically I would have to limit the scope of TLSFilter's ability to intercept TLS traffic to that of traffic on common TLS ports (HTTPS on 443, POPS on 995, etc.), which would naturally allow insecure TLS connections

via non-standard TCP ports;

- TPROXY cannot intercept traffic that originates from the same host, so this entirely prevents the direct use of TLSFilter on client and server endpoints;
- and TPROXY actually rewrites the contents IP packet headers, which I would have to manually reverse.

These were compromises that I did not feel could be justified, so I persevered with the lower-level approach.

4.1.4 Processing Pipeline

Figure 4.1 illustrates a simplified, high-level view of the processes carried out by TLSFilter to reach verdicts for IP datagrams. Verdicts can be one of three possible states: negative or positive, if a final decision has been reached for all IP datagrams of a particular TLS connection, or deferred (temporarily allowed).

The processing pipeline is currently single-threaded but it lends itself well to being extended to exploit parallelism. The processing of callback functions registered by plugins could also be divided among an arbitrary number of threads, for use cases where performance may be bottlenecked by a single processor core.

4.1.5 Initialisation & Tear-Down

On running TLSFilter the following high-level processes occur:

1. The custom kernel module is loaded that redirects all IPv4 traffic to a Netfilter queue
2. All plugins (shared objects) referenced in the configuration file are loaded and initialised
3. The logging system (syslog) is initialised
4. TLSFilter attaches to the Netfilter queue, and issues a verdict after processing each IP datagram

The tear-down process occurs in reverse, but is only triggered in the event of an exceptional failure with Netfilter; under normal circumstances TLSFilter will run indefinitely. In the event of abrupt and/or unexpected termination—for example, if TLSFilter is sent a `SIGKILL` signal—the kernel module will remain loaded and thus packets will be denied until TLSFilter is restarted and resumes issuing verdicts. If TLSFilter is not restarted before the Netfilter queue becomes full, network packets will be rejected, ensuring that no packet is ever mistakenly forwarded without first passing through TLSFilter.

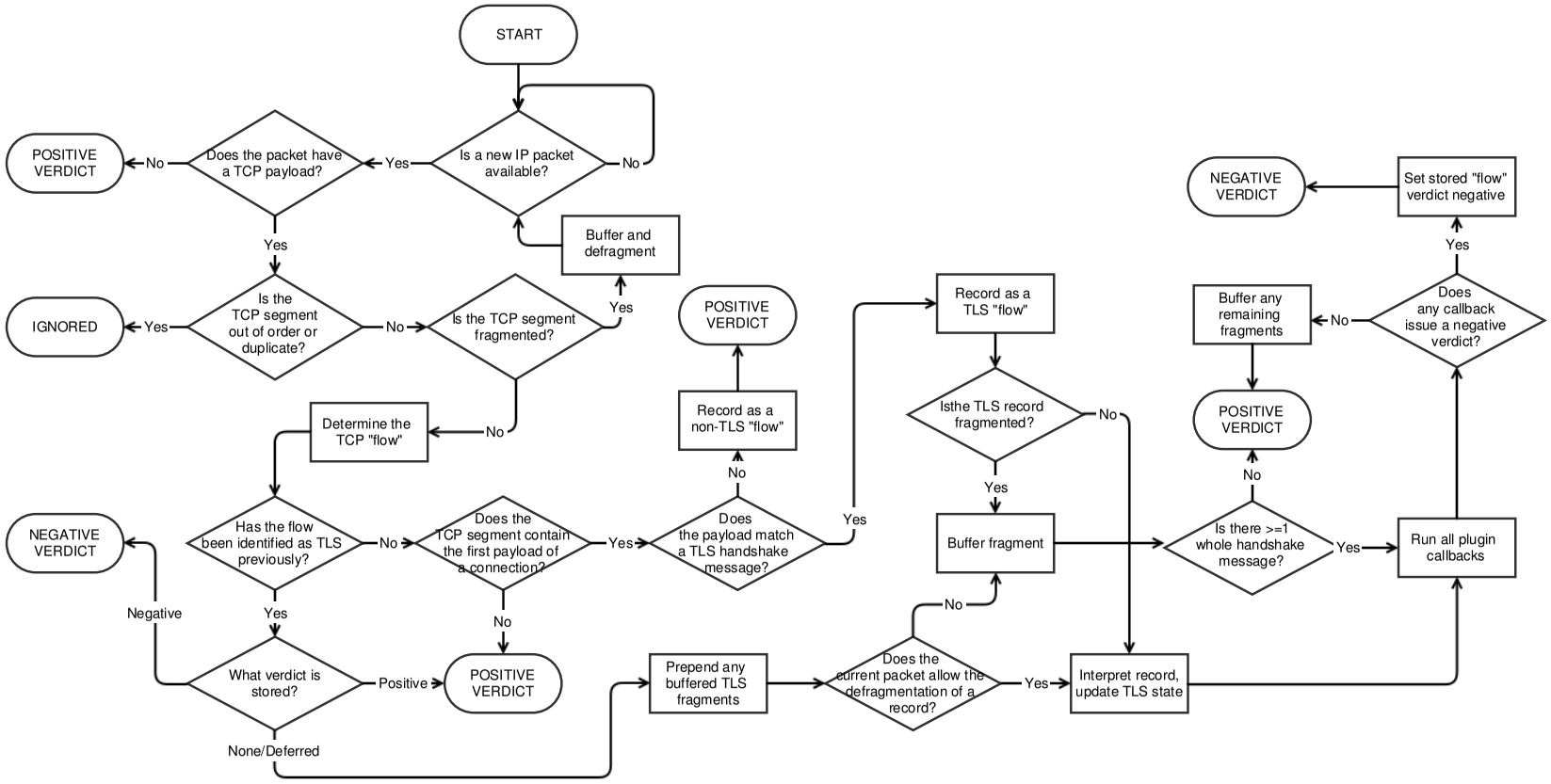
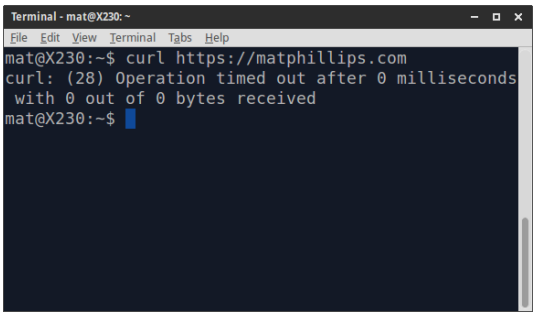
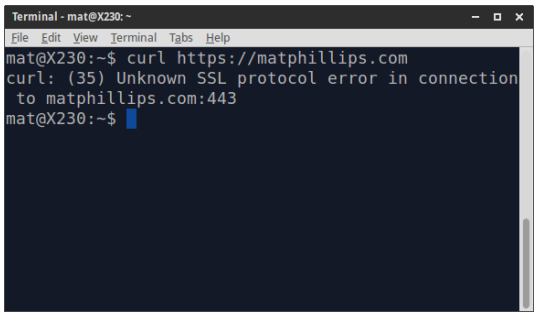


Figure 4.1: Overview of TLSFilter IP datagram processing



(a) Delayed 'soft' failure, without spoofed TCP RST



(b) Immediate 'hard' failure, with spoofed TCP RST

Figure 4.2: A comparison of application-layer behaviour as a result of the `spoof-rst` option

4.2 Usability Concerns

TLSFilter is designed to be executed as a background process that runs indefinitely, with user interaction performed exclusively through the tlsfilter.conf configuration file and log output. Rules are designed to be human-readable and intuitive, and syslog is used as standard for logging events/errors as well as the standard error output stream (if configured accordingly in the configuration file). Additionally, the C plugin API (Section 3.3) is designed to be accessible to anyone with basic proficiency in C.

4.2.1 Hard versus Soft Failure

An unexpected usability issue I experienced after extended use of TLSFilter was the 'soft' failure of TLS connections. Initially, TLSFilter was configured to deny *all* IP packets corresponding to a TCP stream once a negative verdict was issued for a TLS connection, but this caused applications to fail slowly and ostensibly due to a TCP timeout. I made modifications to allow transmission of forged TCP reset (RST) packets, which immediately end the TCP connection and from a usability standpoint make it much clearer that a connection has failed.

From a technical standpoint, this also prevents the TCP/IP stacks on both ends of the connection from attempting to request the retransmission of packets. Technically, forging TCP reset packets is an abuse of the TCP specification, so I have made this feature a configuration option that defaults to being disabled rather than hard-coding it into the source code. To enable this option add `tlsfilter spoof-rst` to the TLSFilter configuration file.

Figure 4.2 illustrates the change in application-layer failure mode when a TLS connection to an insecure server is attempted and the `spoof-rst` configuration option is enabled. The 'soft' failure occurred after a minute or so delay whereas the 'hard' failure returned almost instantaneously.

Chapter 5

Implementation and Testing

5.1 Language Choice

At the start of the project my criteria for evaluating implementation languages for TLSFilter were as follows:

- Easy to write low-level code
- Well supported for Linux kernel interaction
- Well supported for network programming
- General purpose and unrestricted
- Simple
- (Ideally performant)

I identified C, C++ and Go as potential candidates.

Given the the Linux kernel is written in C, I knew that choosing it represented the simplest approach to interaction with the existing Netfilter and networking subsystem, but I was concerned that my speed of development would suffer from the lack of inbuilt library functions and my general lack of experience with lower-level languages.

C++ offered a viable alternative with useful abstractions, including object orientation, and I was aware that the Boost libraries standardised a greater corpus of functionality than the equivalents for C. However, C++ is arguably less well supported by my target environment of Linux, which had (somewhat marginal) potential to discourage adoption and further development.

Go had always interested me as an alternative to C that offered more functionality, including hash tables as a primitive type, as standard. I had concerns as to how simple interaction with

Netfilter would be and the unfamiliarity of the build process since this would require additional compiler software to be installed by end users. I came to the conclusion that these risks did not outweigh the benefits so decided to discount this option.

In summary, I made the conservative choice of C, reasoning that I could switch to C++ relatively easily if circumstances required a greater level of abstraction.

5.1.1 Library Choices

From my background reading I had noticed the trend of complex specifications and software leading to insecurity, so I wanted to focus on reducing unnecessary complexity in TLSFilter as much as possible. To achieve this, I decided that I wanted my source code to be clean and simple and that I could ensure this by being able to justify the execution of every function call. Hence, I took the general approach to keep reliance on external libraries to a minimum.

libnetfilter_queue

libnetfilter_queue is a library that provides an API to Netfilter to allow user space programs to process network packets. I realised quickly that writing and testing Linux kernel modules was very restrictive and that my initial plan to write TLSFilter wholly in kernel space was unrealistic, so a library like libnetfilter_queue became a necessity.

When writing kernel modules, there is no access to shared objects nor the C standard library so everything must be written from scratch or already exist somewhere within the kernel. During my brief stint in kernel module development, I found strange behaviour and race conditions that would cause kernel panics when I allocated memory too frequently. Moreover, debugging a loaded kernel module was a nightmarishly complex process and I ended up resorting to using a virtual machine for development.

libnetfilter_queue allowed me to write the vast majority of TLSFilter in user space, at the performance penalty of having to copy every network packet into user memory and back again. Informal testing did not show a noticeable degradation in performance, so this was a minor compromise for comparatively large gains in development environment sanity and source code simplicity.

Libnids

Libnids was introduced in Section 2.5.2 and its simplicity is well suited to performing a single important purpose in TLSFilter: TCP stream reconstruction. After fixing compatibility bugs with the current Linux kernel, I took the liberty of removing dependencies on other libraries and all functionality not absolutely necessary, which resulted a very thin layer around the Linux TCP stream reconstruction code.

Libnids is built as the static library `libnids.a` and then statically linked to TLSFilter as part of the build process, to avoid the (rare) situation where a user has a version of Libnids lacking my

changes already installed as a shared object, which would cause TLSFilter to fail at runtime.

libssl & libcrypto

libssl and libcrypto are libraries provided by the OpenSSL project for the purposes of SSL/TLS protocol parsing and performing cryptographic operations, respectively. To be clear, TLSFilter has a TLS protocol implementation written from scratch and does not rely upon OpenSSL for any of the SSL/TLS protocol specifics; libssl is used only to parse X.509 certificate binaries and libcrypto is used only to extract public key properties and generate message digests.

GNOME GLib

I used the Gmodule functionality from Glib to support the dynamic loading of shared objects (i.e, custom TLSFilter plugins) at runtime. This means that a user can write a C plugin, compile it to a shared object and then simply reference it in the TLSFilter configuration file and it will be loaded at runtime, with no requirement to recompile the core binary.

uthash.h

uthash provides a very light-weight, entirely macro-based library that implements hash tables that can be indexed by *arbitrary* C data types. Lookups, insertions and deletions are all constant time operations and there is very little memory overhead. I originally found this library when looking for implementations that were usable in kernel space, but its simplicity convinced me to use it throughout TLSFilter.

5.2 Code Portability

With regards to endianness, "network order" is defined to be big endian and the TLS specification does not contradict this. This means that on little endian systems (such as x86 and AMD64) the byte ordering of values extracted from the various data structures used as part of the TLS protocol must be reversed before decisions are made on the basis of these values. To maintain portable code, efforts have been made to use the network-to-host helper functions throughout TLSFilter: `ntohl()` for 32-bit integers and `ntohs()` for 16-bit.

However, unaligned memory accesses are used throughout as a consequence of casting pointers to TLS-specific structs. This has significant impact on the architectures that TLSFilter supports; for example, ARM processors will create an exception interrupt, making use on this architecture impossible. Unaligned accesses also have performance implications for other architectures including x86, as values may need to be (transparently) copied to a word or byte boundary in memory before being accessible.

A alternative approach to struct pointer casting is to write a parser procedure for each wire-format data structure and populate an intermediate logical data structure that has the fields that we are interested in. This approach was not taken because it was too much of a distraction from the overall objectives the project, that is, multiple architecture support—while it is advantageous—does not supersede having a concise and clear implementation.

Since TLSFilter is coupled very tightly to Netfilter, it is not possible to support operating systems that are not based upon Linux.

5.3 Data Structures and Algorithms Choices

5.3.1 Wire-Format Structs

As previously mentioned, TLSFilter makes use of pointer casting to custom structs to parse TLS payloads. One disadvantage of this approach is that C structs cannot contain variable length fields in between fixed length ones, so one logical TLS message may require being split into multiple TLSFilter structs. An example of the structs used to parse ServerHello messages is shown below:

```
typedef struct __attribute__((__packed__)) tls_hs_server_hello {
    uint16_t server_version;       // Highest SSL/TLS version supported
    uint8_t server_random_time[4]; // UNIX time since Jan 1 1970 according to server
    uint8_t server_random[28];     // 28 bytes of (pseudo)randomness from the server
    uint8_t session_id_length;     // Size of the session ID (maximum 32 bytes)
    void * data;           // The variable length session id, followed by everything else
} tls_hs_server_hello;

typedef struct __attribute__((__packed__)) tls_hs_server_hello_tail {
    uint16_t cipher_suite;         // Chosen cipher suite for communication
    uint8_t compression_method;    // Chosen compression method for communication
} tls_hs_server_hello_tail;
```

So to read the compression method field given a pointer to a TLS record, for example, the following must take place:

1. Create a `tls_hs_server_hello` pointer `p1`, by casting a given TLS record payload pointer
2. Create a `tls_hs_server_hello_tail` pointer `p2` and assign it to point to `&(p->data) + p->session_id_length`
3. Now read `p2->compression_method`

One other detail to note is the use of the `__attribute__((__packed__))` compiler tag in the declaration of the structs, which forces the compiler to 'pack' the contents rather than inserting gaps in memory between the fields such that they are aligned to word/byte boundaries.

Unlike if the TLS records were parsed to an intermediate data structure, it should be clear to see that the TLSFilter source code closely follows the wire-format specification from page 27 of the official SSL RFC:

```
...
struct {
    ProtocolVersion server_version;
    Random random;
    SessionID session_id;
    CipherSuite cipher_suite;
    CompressionMethod compression_method;
} ServerHello;
...
```

5.3.2 TLS Flows

A data structure that is used heavily throughout TLSFilter—and supplied to give context to plugin callbacks—is the tlsflow struct. This struct is used to store TLS state information, TLS record fragments and useful lower-level information such as source/destination IP address:port pairs. In aggregation I refer to this information as a 'TLS flow,' similar to how state information for a single TCP connection is sometimes referred to as a TCP flow.

Instances of tlsflow structs are stored in a global hash table, using a simple custom hash function to convert the IP addresses and ports associated with a TCP connection to a 32-bit integer. As this code is executed for every packet received, it needs to be highly performant, so I chose a hash function that uses only the exclusive-or operator, two multiplications and a single 16-bit shift.

```
uint32_t tlsflow_hash (uint32_t addr1, uint32_t addr2, uint32_t src_port, uint32_t dst_port){
    return ((addr2 * 59) ^ addr1 ^ (src_port << 16) ^ dst_port);
}
```

Since the port values will always be $< 2^{16}$, shifting `src_port` losslessly distributes the data across the 32-bit output. As IP addresses are 32-bits there cannot be a a lossless redistribution when shifting, so by multiplying by the prime 59 instead, we redistribute the data in such a way that `addr3` $*59$ has a better chance of being unique than by multiplying by a non-prime. We could remove this multiplication entirely, but it would mean that TCP connections between clients and servers in the same IP subnet are more likely to result in a hash collision due to the loss of asymmetry, since the first few octets are identical.

5.3.3 Constant-time Lookups

While I have been clear from the outset of the project that a focus has not been performance, I should clarify that this is not a focus on *absolute* performance. Since TLSFilter inspects every

IP packet, it has potential to become a network bottleneck, so I have used constant-time lookups through hash tables where possible.

The `tlsflow_hash` function from Section 5.3.2 used for connection tracking is an example of a function that executes in constant time. A similarly frequently-executed section of code determines the applicable rules for a particular IP address. During the initial configuration loading, TLSFilter binds a set of configuration rules to an IP scope, which is consequently bound to callback functions by the plugins in the configuration. When a TLS record is processed, the IP address is checked against a hash table to determine if the particular plugin has any callbacks to be applied in this instance.

Given that the IPv4 address space is large, memory consumption will be high if scopes contain large IP subnets. However, this is at odds with the use cases of TLSFilter: users will want to apply rules globally—meaning that no scope needs to be stored—or specifically to particular TLS instances. It's highly unlikely that a single configuration is applicable to a large enough number of IP addresses such that memory consumption is an issue.

In the certificate plugin, I defined a constant-time lookup-table for the SHA-1 hashes corresponding to weak Debian public keys, so that checking every X.509 certificate received against all 1,179,648 weak keys does not impact performance. The obvious drawback to this approach is also memory consumption, but the scale of the memory consumed here is orders of magnitude lower than the potential memory consumption of IP address to ruleset bindings.

5.3.4 Configuration

The header file `config.h` allows the user, or a script, to tweak memory consumption for their particular use case before compilation. It also defines a number of constants that may require changing as the TLS feature set evolves.

5.4 Low-Level Testing

5.4.1 Runtime Debugging

During general development I configured `gcc` to add debug symbols to the `tlsfilter` binary using the `-g` parameter. This allowed me to use the `gdb` debugger to get extra context when my code hit unexpected fatal exceptions; it also allowed me to step through the code and arbitrarily print the state. Having little previous C experience, this was invaluable for understanding the nuances of how `gcc` interpreted my code; through `gdb` I was able to efficiently find and fix issues caused by struct packing (Section 5.3.1) and unaligned pointer arithmetic (Section 5.2).

Some memory management-based bugs would only occur after extensive use, so I also used Valgrind—notable for being used mistakenly to weaken the PRNG in the Debian-Openssl debable—to identify the causes of memory leaks and rare invalid accesses.

5.4.2 Diagnostic Logging

For debugging errors that were of a more high-level nature, particularly those relating to TLS record fragmentation and the TLS heartbeat protocol, I littered the code with calls to the `LOG_DBG` macro (Section 5.6.2) that wraps a call to `syslog`. As debug messages were frequent and verbose, I required a global configuration option to enable them and another to duplicate their printing to the standard error output stream.

5.4.3 Kernel Module Debugging

Kernel module debugging was *significantly* more difficult than I had expected. To debug a module with `gdb`, the kernel must have been compiled with debug symbols and the `/proc/kcore` virtual file that represents the memory available to the computer must exist. (If not, one can obtain the appropriate branch of the kernel source code and recompile with the `CONFIG_DEBUG_INFO` and `CONFIG_PROC_KCORE` configuration flags.)

Once a kernel with the appropriate features is running, `gdb` must be attached to `vmlinux /proc/kcore` (i.e., the kernel binary with the system memory as an argument) which will consequently allow the reading of variable values but, as the kernel is currently orchestrating the operating system running the debugger, not the low-level stepping through of instructions.

Calls to `printk()` directly in the source code will log information to `syslog` that is then usually written to disk, which is useful if unfixed bugs cause kernel panics which halt the kernel and hence the ability to fix bugs. Much to my annoyance, this happened frequently when I issued a large numbers memory allocation calls, so I abandoned kernel development for the relative comfort of `gdb` in user space.

5.5 High-Level Testing

5.5.1 Testing SSL/TLS Connections

Testing the effectiveness of the firewall required a suite of SSL/TLS instances with varied configurations that imitate the possible servers clients would interact with as a matter of course. To create such a suite of instances, I ran an HTTP server on a single local port and then used `stunnel` to create SSL and TLS listeners that simply handled the encryption layer and forwarded application data to the HTTP server.

`stunnel`'s website describes itself as *"an SSL encryption wrapper between remote client and local (inetd-startable) or remote server... Stunnel uses the OpenSSL library for cryptography, so it supports whatever cryptographic algorithms are compiled into the library."* It makes it very easy to define the precise configuration parameters of a TLS server and to a greater degree than many web server implementations.

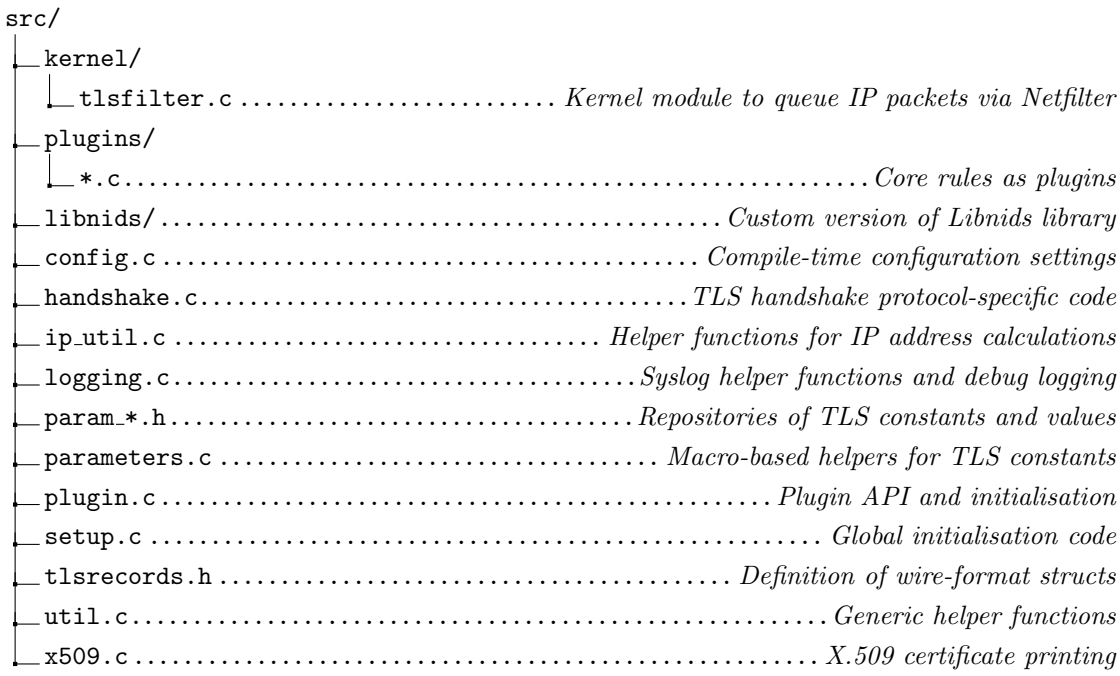


Figure 5.1: Purposes of important source code files

I generated multiple types of public key (DSA, RSA, ECDSA) and then created certificates using the `openssl x509` functionality. Connections were then tested with the `s_client` option of the `openssl` binary, which will attempt a connection to a TLS instance and print diagnostic information if successful.

5.6 Source Code

5.6.1 Overall Structure

Figure 5.1 defines the purposes of important source code files and illustrates the relatively flat structure of the source code.

5.6.2 Use of Preprocessor Tricks

The C preprocessor performs modifications to source code before the compiler sees it, so "code to generate code" can be written to create efficient code with as few runtime checks as possible. I used the preprocessor throughout TLSFilter to make the code cleaner and to reduce duplication.

String Helper

String functions in C are notorious for their ability to allow programmers to introduce vulnerabilities and, as they are not a primitive type, they require—in my opinion—overly opaque and verbose processing code. Since the TLSFilter configuration code required performing a lot of string comparisons, I created the `STRING_EQUALS` macro function to abstract away the underlying code required to compare two strings. The alternative approach would be to create a standard C helper function but that introduces more code and requires the compiler to recognise that the function call can be inlined to achieve the same efficient binary.

Unaligned Pointer Arithmetic

I also frequently needed to perform pointer arithmetic that did not necessarily fall along word/byte boundaries, so to get the compiler to do exactly what I wanted required a litany of ugly casting and bracketing. Moving this to a macro function `UNALIGNED_POINTER_ADD` both reduced duplication and gave some context to purpose of the code, which wasn't immediately obvious otherwise.

Macro Enums

TLS requires the specification and enumeration of names and values for various parameters including supported cipher suites, named elliptic curves and digest algorithms. Given that these get added (e.g., TLS ECC cipher suites) and removed (e.g., 40-bit DES cipher suites) over time, I wanted it to be very simple to update these in response to official changes by IANA.

Accordingly, rather than defining these as C enums, I defined an `ENUM` macro to which I instantiated values in separate header files (see Figure 5.2) allowing these to be updated regardless of C programming proficiency. I then could redefine the `ENUM` macro on the basis of my application, using the the `#include` preprocessor directive in context (see Figure 5.3). This allowed me to write efficient helper functions that were fully defined at compile time with no data duplication whatsoever.

Vararg Logging Macros

Like C functions, preprocessor macro functions can also be defined to take a variable number of arguments. I used this approach to define the various logging macros `LOG_FATAL`, `LOG_ERROR`, `LOG_WARN`, `LOG_DBG` as the underlying `syslog` calls take a format string and an arbitrary number of parameters.

5.7 Build

TLSFilter is compiled using the standard UNIX `./configure && make` convention, where the configure script checks that the shared object dependencies are installed and Make recursively builds

```
ENUM(SECT163K1, 1)
ENUM(SECT163R1, 2)
ENUM(SECT163R2, 3)
ENUM(SECT193R1, 4)
...
ENUM(BRAINPOOLP384R1, 27)
ENUM(BRAINPOOLP512R1, 28)
ENUM(ARBITRARY_EXPLICIT_PRIME_CURVES, 0xFF01)
ENUM(ARBITRARY_EXPLICIT_CHAR2_CURVES, 0xFF02)
```

Figure 5.2: Contents of param_ec_named_curve.h

```
#undef ENUM
#define ENUM(ID, VAL) case VAL: return #ID;

char * curve_to_str(ec_named_curve c){
    switch(c){
#include "param_ec_named_curve.h"
        default: return "UNKNOWN";
    }
}

#undef ENUM
#define ENUM(ID, VAL) if (STRING_EQUALS(s,#ID)) { return VAL; }

ec_named_curve str_to_curve(char * s) {
#include "param_ec_named_curve.h"
    return UNKNOWN_EC_NAMED_CURVE;
}
```

Figure 5.3: Two string helpers from parameters.c that redefine the `ENUM` macro

the project. The configure script was built using the GNU Autoconf tool and the makefile located in the root directory was written from scratch.

The makefile has several targets:

- **kernel**: recursively executes the Makefile in the src/kernel directory and copies the resulting binary to the root
- **libnids**: builds the modified libnids library as static library `libnids.a`
- **tlsfilter**: builds the `tlsfilter` binary from its dependencies
- **plugins**: recursively builds the plugins as shared objects in the src/plugins subdirectory

If no target is explicitly specified as an argument to `make`, the makefile will build all targets in the order specified above.

5.7.1 Compiler Flags

The makefile specifies several non-standard `gcc` flags that are used to improve runtime performance, reduce code size and/or harden security.

- `-O3`: Optimises by reducing execution time, though may do so at the expense of binary size
- `-D_FORTIFY_SOURCE=2`: Replaces unlimited length buffers with fixed length ones, where possible, to improve security
- `-fstack-protector`: Forces the program to abort when a buffer overflow has occurred on the stack
- `-fPIE -pie`: Creates a binary capable of being executed regardless of its absolute position in memory—a "position-independent executable"—so that supported operating systems may use ASLR[1]
- `-Wformat -Wformat-security`: Displays warnings whenever format strings are used in insecure ways

In contrast to my development environment, the `-g` flag has been excluded as storing debug symbols is no longer necessary. Although exclusion of this flag won't result in any runtime performance benefit when compiling with `gcc`, it will reduce the size of the resulting binary.

Along with the security benefits of using these compiler flags, compiling with `gcc` 4.8.2 on 64-bit Ubuntu 14.04 with all aforementioned compiler flags enabled results in a binary size reduction of 33% (from 350 KB to 232 KB).

[1] Address space layout randomisation is a technique where an attacker is prevented from jumping to known positions in memory as a result of buffer overflows

Chapter 6

Evaluation

6.1 Security Analysis

In this section I attempt to quantitatively analyse any security benefits gained by using TLSFilter, relative to an attack model with three adversaries that have varied levels of skill, access to capabilities and knowledge. Although I believe this represents a thorough analysis of likely attacks against TLS, it would be dangerous to assume it complete.

6.1.1 Schneier Attack Tree

Figure 6.1 shows a Schneier attack tree[37], which describes the possible chains of attacks that can be used to achieve the goal of reading the plaintext of an encrypted TLS connection. Nodes are designated with "OR" and "AND" to show whether one or all of the child nodes are required for the attack represented by the node to be satisfied.

In my attack tree, leaf nodes are assigned a probability based on the characteristics of the TLS connection and the profile of the adversary, which is combined and propagated up to the root node. The value at the root allows the comparison between combinations of targets, adversaries and TLS configurations. For simplicity, it is assumed that the value of an "AND" node is the product of all child node values and the value for an "OR" node is the maximum of all child values (i.e., it is assumed the adversary will pick the optimal attack).

For clarity of attack, I have split some nodes into a chain, but this can be thought of as an "AND" node with all chain members as leaf nodes. Any node with an asterisk is a repeated node that is defined elsewhere in the tree.

This attack model has been modified through the removal of nodes to take into account TLSFilter's feature set, with the resultant attack tree shown in Figure 6.2.

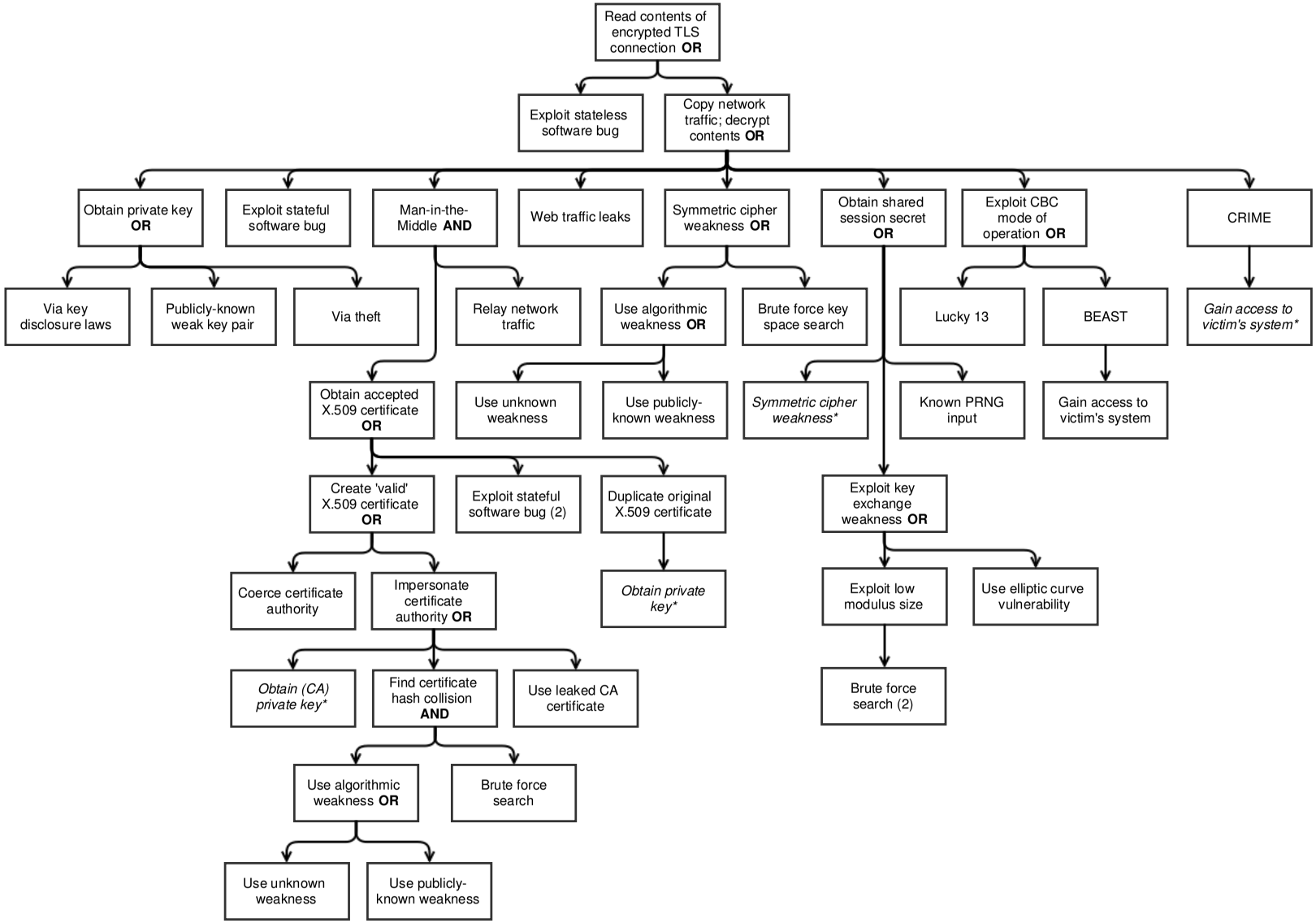


Figure 6.1: Schneier attack tree with the goal to read the contents of an encrypted TLS connection

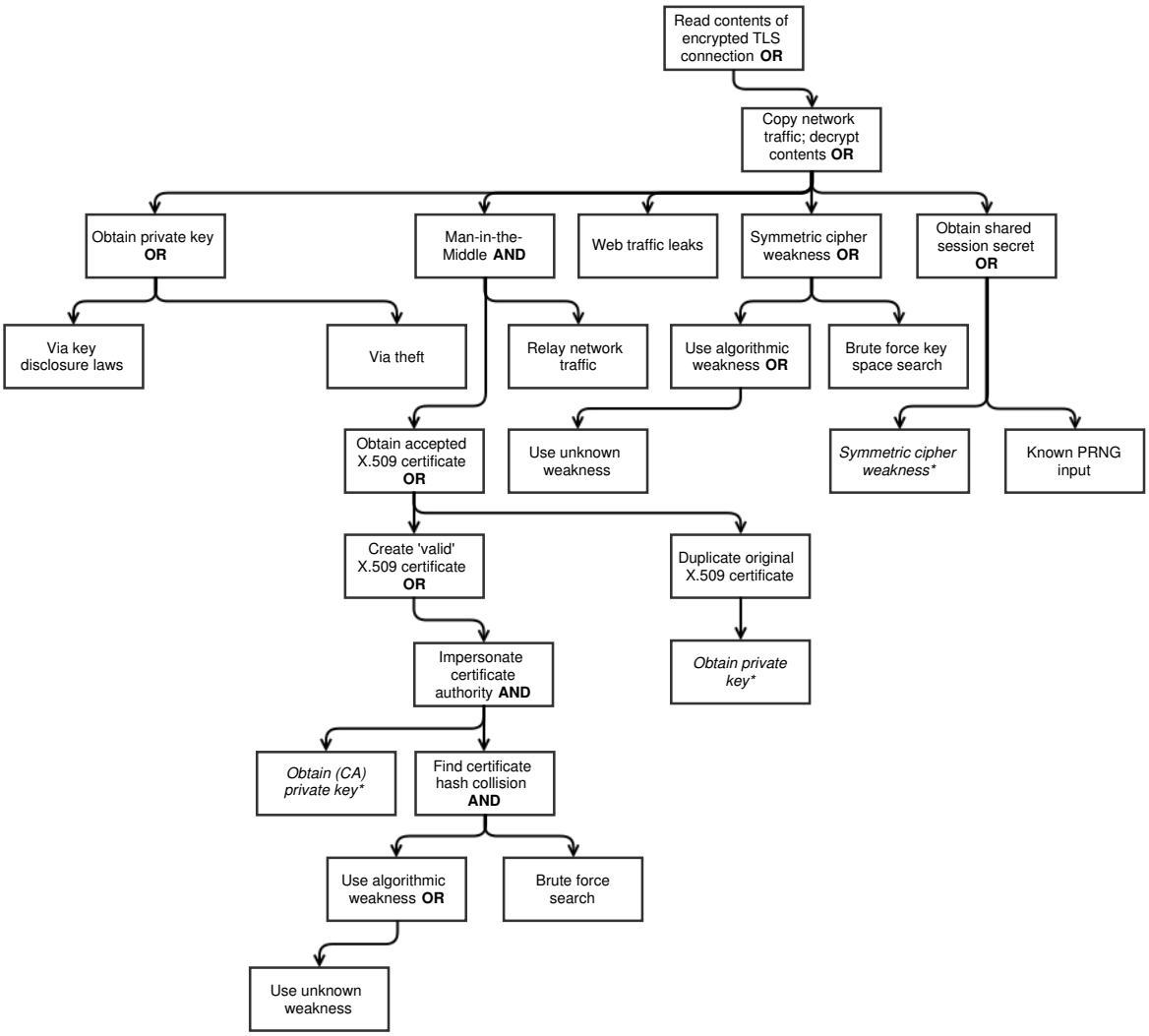


Figure 6.2: Schneier attack tree with the goal to read the contents of an encrypted TLS connection, under the protection of TLSFilter

6.1.2 Adversary Profiles

The following section describes the three types of adversary used in my security analysis of TLS-Filter. I have assumed that the goal of these adversaries is to read the contents of an encrypted TLS connection and that they have, at minimum, the capability to monitor network traffic. To reduce dimensionality, I am restricting the scope of my attack modelling such that it is assumed the victim's computer is secure.

	Nation State	Criminal Gang	Lone Cracker
Skill level	High	High	Medium
Computational power	High	Medium	Low
Access to zero-day software exploits[1]	High	Medium	None
Access to unknown cryptographic weaknesses	High	None	None
Ability to forge network traffic	High	Low	None
Risk appetite	Low	Medium	Medium

Nation State

A large group of highly-skilled and well-resourced technical professionals. Assumed to have resources including significant computing power and access to global network backbones. Likely to have privileged knowledge of cryptographic weaknesses and numerous zero-day software implementation exploits. Operates clandestinely and is largely risk averse.

Criminal Gang

A small group of people with a wide breadth of technical skills and similar goals. Likely to have access to transient resources via botnets and may have knowledge of a small number of zero-day software implementation exploits, but a large arsenal of tools to exploit public vulnerabilities. Broadly well-connected to other criminal activity and motivated largely by profit.

Lone Cracker

A single person with strong, but perhaps narrow, technical knowledge. Has very little access to computational power, no access to zero-day bugs, but good knowledge of public vulnerabilities and a high propensity to exploit them using freely-available tools. More likely to be motivated by ideology than profit.

6.1.3 Attack Tree Instances

The probabilities for the attacks in the tree depend on the nuances of the victim's TLS configuration as well as the attributes of the victim and the motivations of the adversary. This high dimensionality

means is beyond the scope of this project to perform analysis for the general case, so I have chosen three representative targets that are valuable to all three adversaries:

1. ESET; `secure.eset.co.uk:443`; security vendor; creators of the NOD32 anti-virus software
2. Lavabit; `lavabit.com:443`; (defunct) privacy-preserving email provider
3. First Look Media; `firstlook.org:443`; news organisation specialising in investigative journalism

I have made probability estimates for the values of each attack tree leaf node, for each adversary. By consequently applying the propagation methodology mentioned previously to the root, we obtain the results in Table 6.1 (see appendix Tables B.1, B.2 and B.3 for full details). These results show that—under the attack tree model defined earlier, for the three specific cases—TLSFilter always increases the difficulty of an attack.

Furthermore, when TLSFilter is used the most likely remaining attacks are *non-technical*: private key theft and use of key disclosure laws, both as a pre-requisite to performing man-in-the-middle attacks. Mandatory use of cipher suites supporting forward secrecy mitigates the threat of retroactive decryption of TLS connections, but would not prevent future man-in-the-middle attacks if key material were not kept private. TLSFilter's support for certificate pinning would do nothing in this case, as there is no possible way to identify a man-in-the-middle attack if the certificate indistinguishable to the original.

In general, I am of the opinion that the "without TLSFilter" results justify the creation and use of TLSFilter. In particular, the unexpected use of weak ephemeral key exchanges, through small moduli and—in my opinion—unjustifiably untrustworthy elliptic curve standards,

Though these results hinge upon my probability estimates, where possible I have attempted to justify these values with external data from Qualys SSL Labs and the NCC ChangeCipherSpec diagnostic tools. I am comfortable in drawing the conclusion that TLSFilter's countermeasures increase the security of TLS connections relative to publicly known threats. Of course, this does not mean that a network protected by TLSFilter can consider its TLS connections to be 'secure', only securer.

6.1.4 Limitations of Analysis

This analysis does not take into account any inadvertent vulnerabilities unwittingly introduced by the use of TLSFilter. My knowledge of C was minimal at the start of this project and it is highly unlikely that I have avoided introducing additional vulnerabilities. Given my background reading, I would argue that the biggest threat to the security of TLS lies in its software implementations rather than its cryptography and, consequently, would recommend against using TLSFilter until it has been independently assessed.

Target	Adversary	Without TLSFilter		With TLSFilter		Reduction
		Score	Attack	Score	Attack	
ESET	Nation state	1.0	export cipher suites	0.2	private key theft	0.8 / 80%
	Criminal gang	1.0	export cipher suites	0.1	private key theft	0.9 / 90%
	Lone cracker	1.0	export cipher suites	0.1	private key theft	0.9 / 90%
First Look Media	Nation state	1.0	ChangeCipherSpec man-in-the-middle	0.4	key disclosure law	0.6 / 60%
	Criminal gang	0.4	weak 1024-bit DHE modulus	0.1	private key theft	0.3 / 75%
	Lone cracker	0.2	weak 1024-bit DHE modulus	0.1	private key theft	0.1 / 50%
Lavabit	Nation state	1.0	ChangeCipherSpec man-in-the-middle	0.8	key disclosure law	0.2 / 20%
	Criminal gang	0.4	weak 1024-bit RSA and DHE moduli	0.1	private key theft	0.3 / 75%
	Lone cracker	0.2	weak 1024-bit RSA and DHE moduli	0.1	private key theft	0.1 / 50%

All node values for these scenarios are available in the appendices.

Table 6.1: Nominal probability values for attack scenarios

In addition, the reliability of my analysis and any conclusions drawn are bounded by the attack tree model and its efficacy is bounded by public knowledge of attacks; in particular, TLSFilter cannot protect against the entire class of unknown implementation bugs. Had my analysis been performed before the publication of the OpenSSL ChangeCipherSpec injection attack (and before the creation a corresponding TLSFilter plugin to mitigate this threat), then my conclusions regarding real-world impact would have been unknowingly flawed.

With regards to my choice of modelling formalism, I modified the Schneier attack tree approach because a basic attack tree would ignore the differing abilities and preferences of possible adversaries. Moreover, given the ubiquity of TLS, there are multiple types of attackers and targets that must be analysed, so using a basic Schenier attack tree would be an oversimplification.

An alternative was to use the Adversary View Security Evaluation (ADVISE) formalism[30], which better quantifies differences in the knowledge, access, skills and goals of attackers. However, for this approach to yield a reliable answer the probabilities for each property of attacker instance must also be reliable, but my figures are necessarily speculative. My modified approach, where each attack tree leaf node has an overall probability fixed in relation to the attacker and target, allows the dimensionality of speculation to be minimised to single values in the hope that this yields a more reliable result.

6.2 Impact on Reachability

The binary nature of TLSFilter's logic—i.e., communication occurs securely or not at all—is, in my opinion, its largest drawback. If every mitigation TLSFilter supported were enabled and applied globally, the reachability issues network users would experience may in fact not justify its use. So, naturally, the side effect of mitigating some types of attack is an increased occurrence of false positives; frequently the TLS parameters that are mutually preferred between a client and server are insecure and TLSFilter will simply block the connection. Unfortunately, this means there is an unavoidable trade-off between TLS attack mitigation and network reachability.

Through my own usage of TLSFilter, I noticed that globally enabling all attack mitigation techniques gave me significant reachability issues, to the extent that normal web browsing was no longer practical. My usage shifted towards customising a few IP subnets with very high security and then configuring a global security 'baseline' that universally improved security but not to the extent that reachability was impacted too significantly.

To find out an optimal baseline configuration, suited specifically to web browsing, I decided to perform some analysis on the TLS configurations of the Internet's most trafficked websites. I then used this data to evaluate the impact on reachability resulting from specific TLSFilter rules and consequently derived a default configuration that I claim offers an acceptable balance between security and reachability.

Two TLSFilter rules increase security without any impact on reachability: ChangeCipherSpec

injection prevention and, in the context of web browsing, the disabling of TLS heartbeats. Some rules were also excluded from my analysis because they protect against highly impractical attacks and significantly impact reachability, e.g., disabling CBC cipher suites to prevent Vaudenay padding attacks (Section 2.4.2) and disabling TLS compression to prevent CRIME (Section 2.4.3).

6.2.1 Modelling HTTPS Traffic

In 2001, Adamic and Huberman published a paper about trends and relationships in web browser traffic and concluded that *"the distribution of visitors per site follows a universal power law, implying that a small number of sites command the traffic of a large segment of the Web population."*[35] This means that any conclusions regarding the security gained by using TLSFilter for the most popular websites should be representative of the average user's overall security gain.

In a follow-up paper in 2002, Adamic and Huberman demonstrate that the frequency distribution of unique website visits with respect to website rank can be approximated by a Zipfian distribution with rank exponent of 1.0[36]. That is, the expected number of visitors to a website with rank r is $\propto r^{-1}$. Consequently, we should weight our reachability results to imitate this distribution rather than considering the reachability of all popular websites with equal importance.

6.2.2 Results

Choice of Metrics

In Table 6.2, the "blocking score" for a TLSFilter rule r over N ranked websites is defined as

$$\sum_{i=1}^{N} \left(\text{reachable}_r(i) \times \text{rank}(i)^{-1}\right) \tag{6.1}$$

where

$$\text{reachable}_r(i) = \begin{cases} 1 & \text{if connection is denied under TLSFilter rule } r \\ 0 & \text{otherwise} \end{cases} \tag{6.2}$$

It is also important to note that Zipf's Law tends to hold on average rather than for every data point, which tends to bias the true distribution of highly-ranked values that are deemed progressively disproportionately more 'important' under Zipf's Law. That is, it is unrealistic for the highest ranked website *google.com* to be *exactly twice* as important as the second-ranked website *facebook.com*, exactly three times as important as the third-ranked and so on.

To correct this bias, I have normalised the weights of the top 1000 ranked websites using global web traffic estimates sourced from Alexa. Ideally, this metric should have been used to weight my entire data set but unfortunately—due to the commercial nature of the data—this approach is cost prohibitive.

The "blocking score" percentages should be taken as approximations to the proportion of connections made to TLS endpoints that are blocked by TLSFilter. Table 6.2 shows this as both a percentage of HTTPS/TLS connections and as a percentage of *all* web browsing traffic, to highlight overall impact.

Analysis

In addition to a weighted score, Table 6.2 shows the absolute numbers and proportions of blocked websites which provides a (May 2014) snapshot of the current state of publicly-available TLS configurations.

It is interesting to note that the majority (50.6%) of TLS-supporting websites do not support the latest version, TLS 1.2, which was standardised in 2008. This is closely followed by 49.5% of TLS-supporting websites that support only TLS 1.0 and below. The weighted percentages are 27.6 and 26.1 respectively, demonstrating that the likelihood of an HTTPS connection supporting the newer standards is higher, but still not enough to justify applying the `version deny TLS_1_0` and `version deny TLS_1_1` rules globally.

Weak "export grade" encryption was selected by default on 6,419 of the websites tested, which corresponds to approximately 1.1% of TLS connections. RC4 is unusually prevalent—given its keystream bias weaknesses—and, surprisingly, disproportionately over-represented in the weighted score: 8% of all websites versus 9.2% of all connections. (However, I would still deem this low enough to justify considering denying RC4 in a global TLSFilter configuration.)

The results show very little support for Diffie-Hellman ephemeral key exchanges with moduli greater than 1024 bits, which seems unnecessarily low. Furthermore, denying the use of NIST's P-256 named elliptic curve during elliptic curve ephemeral key exchanges has the greatest impact on reachability of all rules tested. Unfortunately, this drastically limits the usefulness of TLSFilter's forward secrecy support: it's not viable to require forward secrecy for every connection, nor enforce a sensible 2048 bit modulus for ephemeral DH key exchanges, nor deny the suspicious NIST elliptic curves.

I do not believe that this is a weakness of my approach, but rather a symptom of outdated configurations and poor support for strong forward secrecy in software implementations. Ultimately, we are stuck between a rock and a hard place, where no trusted solution exists without significantly impacting usability or security.

6.2.3 Proposed Default Global Configuration

Consequently, the following TLSFilter configuration represents a 'baseline' level of security that I would recommend to be applied globally:

```
version deny_SSL_3_0
parameter deny cipher null
```

	Blocked Websites			Blocking Score	
TLSFilter Rule	Frequency	% of Total	% of TLS Total	% of Max.	% of TLS Max.
`keyexchange deny named-elliptic-curve secp224r1`	0	0.0	0.0	0.0	0.0
`keyexchange deny named-elliptic-curve sect233r1`	0	0.0	0.0	0.0	0.0
`keyexchange deny named-elliptic-curve sect283r1`	0	0.0	0.0	0.0	0.0
`parameter deny cipher rc2`	0	0.0	0.0	0.0	0.0
`keyexchange key-length-minimum dh 512`	2	0.0	0.0	0.0	0.0
`parameter deny cipher null_cipher`	5	0.0	0.0	0.0	0.0
`keyexchange key-length-minimum dh 768`	15	0.0	0.0	0.0	0.0
`keyexchange deny named-elliptic-curve sect163r2`	16	0.0	0.0	0.0	0.0
`certificate key-length-minimum * 512`	37	0.0	0.0	0.0	0.0
`parameter deny cipher idea`	57	0.0	0.0	0.0	0.0
`parameter deny cipher seed`	115	0.0	0.0	0.0	0.0
`certificate key-length-minimum * 768`	147	0.0	0.1	0.0	0.0
`certificate key-length-minimum * 1024`	158	0.0	0.1	0.0	0.0
`keyexchange deny named-elliptic-curve sect571r1`	168	0.0	0.1	0.0	0.0
`keyexchange key-length-minimum dh 1024`	272	0.1	0.1	0.0	0.0
`keyexchange deny named-elliptic-curve secp521r1`	286	0.1	0.1	0.0	0.0
`keyexchange deny named-elliptic-curve secp384r1`	75	0.0	0.0	0.2	0.3
`version deny SSL_3_0`	2151	0.5	0.9	0.6	0.9
`parameter deny cipher DES`	6247	1.3	2.7	0.7	1.1
`parameter deny export-grade`	6419	1.4	2.7	0.7	1.1
`certificate key-length-minimum * 2048`	28561	6.2	12.2	2.2	3.4
`parameter deny digest TLS_MD5`	4764	1.0	2.0	2.4	3.7
`parameter deny cipher RC4`	37350	8.0	16.0	9.2	14.2
`keyexchange key-length-minimum dh 2048`	95792	20.6	40.9	10.1	15.6
`keyexchange key-length-minimum dh 3072`	97046	20.9	41.5	10.2	15.8
`keyexchange key-length-minimum dh 4096`	97046	20.9	41.5	10.2	15.8
`parameter require forward-secrecy`	75282	16.2	32.2	16.2	25.1
`version deny TLS_1_0`	115881	25.0	49.5	16.9	26.1
`version deny TLS_1_1`	118374	25.5	50.6	17.8	27.6
`parameter deny digest TLS_SHA`	137924	29.7	58.9	20.8	32.2
`keyexchange deny named-elliptic-curve secp256r1`	61123	13.2	26.1	38.0	58.8

Table 6.2: Reachability results for TLSFilter rules for the most popular 464,259 websites, ordered by ascending 'blocking score'

```
parameter deny export-grade
parameter deny digest TLS_MD5
certificate key-length-minimum 2048
keyexchange key-length-minimum dh 1024
ccsmitm deny
heartbeat deny
```

The weighted blocking score as a percentage of total web browsing using this configuration is 5.5% (or 11.8% with `parameter deny cipher RC4` added). Additionally, the total number of websites blocked using this configuration is 40,312; if there were no overlap among the rules we would expect 42,172 to be blocked, suggesting that many insecure websites are insecure for multiple reasons.

6.2.4 Limitations of Analysis

Ignores TLS Client Support

My methodology abstracts away the feature set support of TLS clients and always assumes that a client will support the server's most preferred configuration. Realistically, support for cipher suites, TLS versions and ephemeral key exchange algorithms varies between web browsers, web browser versions, operating systems, TLS libraries and TLS library versions. It is impractical to test for all of these values nor is sufficient data available that breaks down the distribution of TLS clients for each website.

Conversely, my methodology ignores the (flawed) approach that many browsers resort to upon the failure of a TLS connection: re-attempting with alternative preferences. On the balance of probability—and assuming a coherent TLSFilter configuration—connections are more likely to be re-attempted with preferences that reduce security and will consequently blocked by another TLS-Filter rule. Hence I feel justified that my blocking results are not overly inflated.

I also assume that the server is implemented to follow the best practice of prioritising its cipher suite preferences over those of the client. While this is, by far, the most popular approach, if a few highly-ranking websites ignore this practice then the impact on the results would not be insignificant. My analysis should not therefore replace real-world reachability testing and iterative refinement of a global TLSFilter configuration.

The 'Average' User

My methodology is also geared towards an 'average' web user that does not exist. The top 15 websites in my ranking data includes five Chinese-language websites and both US and Indian local Google websites, which are highly unlikely to be visited the proportions suggested by the data by a single user (or even geographical populations of users). An attempt was made to produce

statistics stratified by country, but no reliable source of traffic data could be found to correct the bias introduced by the uniformity of Zipf's Law, so this approach was abandoned.

Geographical Bias

The use of load-balancing (through Anycast IP routing) and content distribution networks means that my results may be biased geographically to France and Moldova, as data was gathered via servers hosted in Roubaix and Chişinău. Whether this has any impact on TLS configurations is unknown.

Reliance on Estimated Data

My conclusions heavily rely on two sets of data which are both supplied by Alexa and hence sampled from the same population of web users that have consented to install the Alexa toolbar and have their usage monitored. Ideally, I would have integrated other third parties' data, including Quantcast, whose methodology tracks (server-side) traffic in a less biased manner, but I could not justify the monetary cost. (Alexa charges per URL through Amazon Web Services, so obtaining traffic data for the top 1000 websites was relatively inexpensive.)

6.3 Performance Testing

To benchmark the performance of TLSFilter, I set up the following on a lab machine:

- `nweb` web server[2]; a performance-optimised web server serving a blank index page and a compressed 100MB file from a RAM-based file system
- `stunnel` to wrap TCP connections to the web server with an SSL/TLS layer, with the default settings and a self-signed certificate

On my local laptop I set up TLSFilter with the following basic configuration:

```
# Global configuration settings
tlsfilter spoof-rst
tlsfilter log-console

parameter deny export-grade
compression disable
certificate deny-weak-rsa-keys
heartbeat deny heartbleed-cleartext
heartbeat disable
```

[2] http://www.ibm.com/developerworks/systems/library/es-nweb/index.html

	x_i					$\bar{x}$	s
With TLSFilter	1.126	1.138	1.132	1.132	1.137	1.135	0.004
	1.137	1.139	1.137	1.134	1.138		
Without TLSFilter	1.123	1.124	1.124	1.124	1.140	1.127	0.0052
	1.126	1.125	1.131	1.124	1.126		

Table 6.3: Time in seconds taken to transfer a 100MB file over gigabit Ethernet

I connected my laptop to the DoC LAN through gigabit Ethernet, presumably on the same switch as the lab machine. All tests were performed on a Saturday afternoon to reduce the likelihood of other network traffic impacting the results.

6.3.1 Throughput

I used the following command to compare network throughput under the presence of TLSFilter: `time wget https://146.169.53.48:8888/100mb.bin -O /dev/null --no-check-certificate`. This command will record the time it takes for `wget` to download and immediately discard the 100MB file served by the lab machine. The results along with the sample mean and standard deviation are shown in Table 6.3.

The results show an overall download speed of approximately 88.1 MB/s when TLSFilter is running, as opposed to 88.8 MB/s without, which represents a 0.78% decrease in throughput. Given that using `time` to gather data will capture the *entire* execution time of `wget`—not just the time spent transferring data over the network—it is likely that these results are lower than real-world values.

These values are also close to the maximum throughput of gigabit Ethernet and, since I am unable to control other network traffic, it is possible that the actual performance TLSFilter is bottlenecked by the network. Accordingly, I do not believe that use of TLSFilter results in any meaningful degradation of network throughput of single connections over gigabit Ethernet.

6.3.2 Latency / Concurrency

As mentioned previously, TLSFilter lacks parallel processing so my intuition was that situations involving multiple concurrent users were the most likely to adversely impact performance.

Throughput alone is not a very representative metric for performance, so I wanted to test connection latency with an emphasis on concurrent connections, as this is a very likely real-world use case. To test this, I ran `nweb` and `stunnel` locally so that external network conditions could not affect my results; there are multiple round-trips in the TLS handshake that cannot be pipelined, so network characteristics and utilisation are notorious for affecting TLS performance.

		Connections			Connection times (ms)			
	Total time (s)	Successful	Failed	Per second	Min	Mean	Median	Max
With TLSFilter	31.17	20000	0	641.66	5	292	218	4622
Without TLSFilter	23.23	20000	0	860.87	4	231	197	3262

Table 6.4: Statistics for 200 concurrent connections

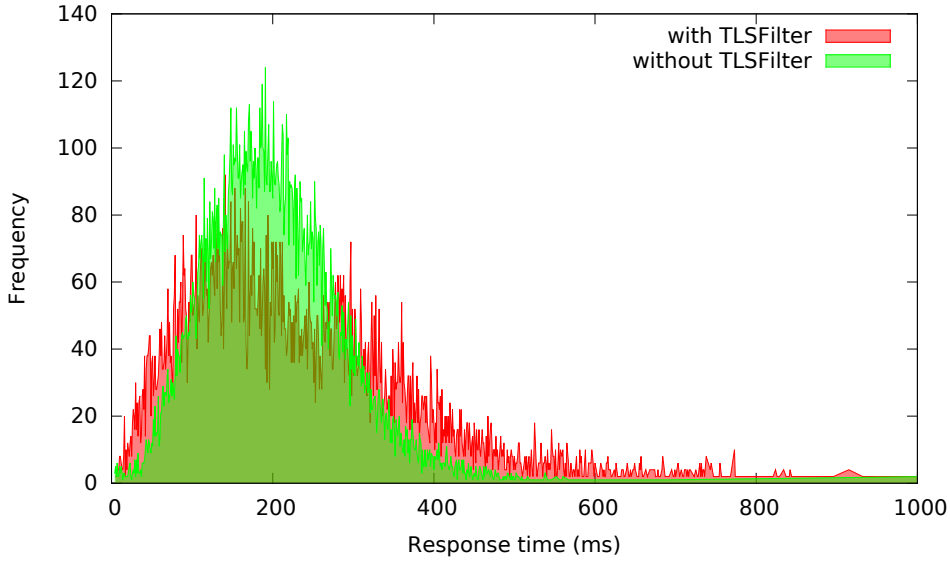


Figure 6.3: Frequency distribution comparison of TLS latency

I used the same configuration from the previous tests but now opting to download the 0B file as opposed to the 100MB file. I also used Apache Bench, which is benchmarking software aimed at web servers but fully supports TLS. My tests were performed with a liberal 200 concurrent connections and 20000 total connections, using the command `ab -n 20000 -c 200 https://127.0.0.1:8888/`. In addition, raw data was saved using the `-g` argument and later used to generate frequency charts with GNUPlot.

The results are shown in Table 6.4. With regards to throughput, using TLSFilter reduces concurrent throughput by 25% from 199.25 KB/s to 148.51 KB/s, which is not insignificant. Plotting the two sets of raw data produced by Apache Bench (see Figure 6.3) reveals a distribution with an interesting and unexpected property: using TLSFilter resulted in a *higher* frequency of responses within the 0-110ms range than the control trial (see Figure 6.4). Even more surprisingly, this behaviour is consistent with repeated attempts.

My intuition was that the data obtained under the influence of TLSFilter would duplicate the

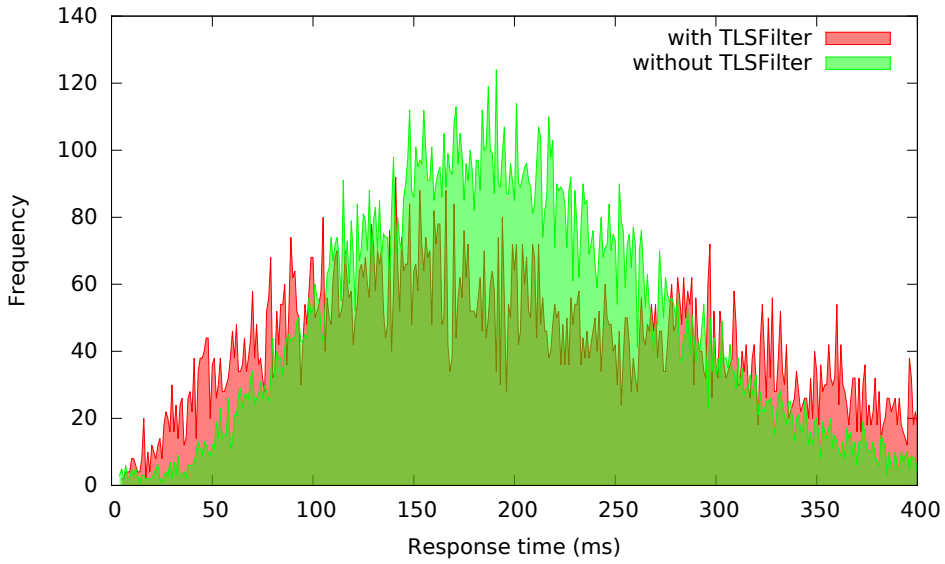


Figure 6.4: Frequency distribution comparison of TLS latency, between 0 and 400ms

distribution of my control trial but with a fixed translation in the positive x direction and, most likely, with increased variance; I assumed that the updated frequency distribution would be bounded by baseline performance. The data shows that this is not precisely what happens, which at first did not make much sense. After a little experimentation, I reasoned that removing packets from the normal Netfilter Linux Kernel pipeline via `libNetfilter_queue` and independently issuing verdicts on packets must been the cause of this unusual performance.

To confirm this hypothesis, I branched the TLSFilter code base and hard-coded a positive verdict for every packet; running this would mean packets would be intercepted via Netfilter and then immediately passed back to the network. I then ran another Apache Bench trial and then plotted the raw data, which supports this hypothesis; moreover, this new distribution better matches what I had expected of a 'baseline' distribution. It would be interesting to find out why it is the case that redirecting IP traffic through `libnetfilter_queue` results in a consistent reduction in the median connection time compared to delegating responsibility to the kernel, but that is beyond the scope of this project.

Table 6.5 now includes the data obtained under the 'branched' implementation and Figures 6.5 and 6.6 include plots of all three distributions.

In general, latency is (expectedly) worse under TLSFilter. However, this does not deviate hugely from the control distribution and is in the imperceptible order of tens of milliseconds. My test configuration is also rather synthetic in that, due to network latency and the number of round-trips

		Connections			Connection times (ms)			
	Total time (s)	Successful	Failed	Per second	Min	Mean	Median	Max
Branched TLSFilter	25.41	20000	0	787.26	3	249	170	4993
With TLSFilter	31.17	20000	0	641.66	5	292	218	4622
Without TLSFilter	23.23	20000	0	860.87	4	231	197	3262

Table 6.5: Statistics for 200 concurrent connections with revised baseline methodology

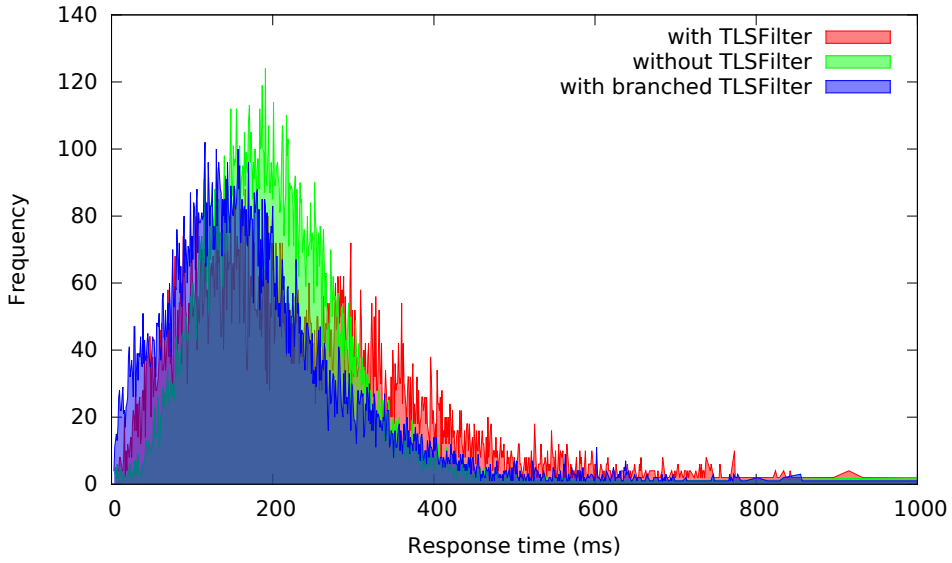


Figure 6.5: Frequency distributions of TLS latency including 'branched' TLSFilter version

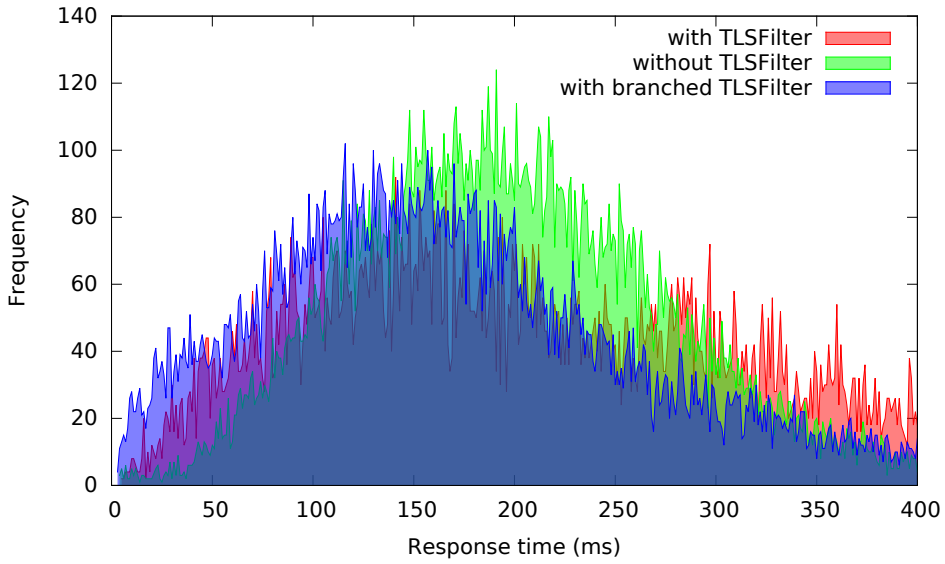


Figure 6.6: Frequency distributions of TLS latency including 'branched' TLSFilter version, between 0 and 400ms

required, a TLS connection over a WAN is highly unlikely to complete within 200ms.

TLSFilter operates serially—albeit giving the impression of concurrency as a result of the packet-switched nature of IP networks—and I am in no doubt that this is the cause of the 25% reduction in throughput. Given that TLSFilter is still able to handle 200 concurrent TLS connections, I do not think this poses a problem to its usage and I believe these results validate my early decision to prioritise the avoidance of complexity over the pursuit of performance.

6.3.3 Limitations of Analysis

Due to an oversight during testing, all measurements were made on an unoptimised version of the TLSFilter binary. As the TLSFilter source code has not changed significantly since these measurements were taken, I claim that these results indicate a minimum level of performance. Given that the updated compiler flags are for both performance and security purposes, it is not clear to me whether these changes will have a positive net effect on performance.

With greater foresight, I could have obtained some 10GbE networking equipment to verify whether TLSFilter was bounded by the gigabit Ethernet hardware used during my testing. I also frequently encountered Linux kernel resource limit errors when testing concurrent connections using Apache Bench, even after ostensibly removing all soft limits. Though 200 connections was a fairly reliable upper limit, it would be interesting to investigate kernel options that would allow me to

better ascertain TLSFilter's true limits.

Ideally I would have written my own SSL/TLS capable server for testing to reduce my dependence on third-party software; it is not clear whether the underlying performance profiles of `stunnel` and `nweb` have introduced biases into my performance measurements, though their reputations for high performance leads me to believe that any performance impact is minimal.

6.4 Summary of Strengths

With regards specifically to my approach, I think TLSFilter offers several advantages over similar software:

- No longer is there sole reliance on application developers nor system administrators to ensure TLS security
- Protects (legacy) embedded systems and network appliances that remain infrequently patched
- Performs deep packet inspection of every IP packet, rather than relying on port numbers or heuristics to naïvely classify traffic
- Mitigates an entire class of fragmentation-based IDS attacks through the use of TCP reassembly logic from the Linux kernel
- Easily extensible, evidenced by the reactive authoring of plugins to mitigate Heartbleed and OpenSSL's ChangeCipherSpec bugs
- Maintains the end-to-end encrypted nature of TLS connections, unlike Blue Coat's SSL Visibility (Section 2.5.3)
- Negligible performance loss with respect to latency and single-threaded throughput
- Immediately deployable
- Simple source code

6.5 Summary of Weaknesses

However, TLSFilter also has a number of weaknesses and limitations:

- Binary decision-making removes the 'negotiation' element of the protocol and introduces a security-reachability trade-off
- No ability for end users to overrule a decision, unlike traditional desktop firewall software and web browser security warnings

- May require manual periodic re-configuration
- Incomplete support for SSL and TLS protocols:
  - Does not attempt to be a full, RFC-compliant implementation:
    - Still reliant on underlying client/server implementations for standard validity checks (e.g., cryptographic operations, X.509 certificate validation, etc.)
    - Will not *proactively* defend against implementation bugs in underlying TLS implementations
  - No support for handshake message fragmentation
  - No support for SSL v2
  - No support for datagram TLS (TLS over UDP)
- May introduce TLSFilter implementation-specific vulnerabilities

Chapter 7

Conclusion

This thesis documents numerous state-of-the-art threats to TLS and consequently constructs an attack model using a modified Schneier attack tree to quantify relative probabilities against disparate targets and adversaries. This research was used to develop a software implementation, TLSFilter, that provides technical mitigations to a variety of TLS architectural weaknesses. In addition, TLSFilter provides an extensible, general-purpose TLS firewall engine upon which future attack mitigation techniques can be built; the unexpected publication of Heartbleed and ChangeCipherSpec injection vulnerabilities provided opportunity to evidence the utility of this functionality.

This thesis analyses 460,000 of the most popular websites and proposes a default configuration that globally improves security for a minimal 5.5% reduction in network reachability. Analysis also shows that (an unoptimised instance of) TLSFilter has a negligible effect on single-connection throughput on gigabit Ethernet, introduces latency in the order of tens of milliseconds and reduces throughput at 200 concurrent connections by approximately 25%.

TLSFilter allows advances in TLS attack mitigation techniques, such as certificate pinning, to be applied universally and irrespective of underlying software implementation, operating system support or network device. It replaces the traditional attack mitigation alternative of waiting for security updates from vendors or manually identifying, patching and recompiling all instances of vulnerable TLS software.

However, TLSFilter is not a panacea as the threats to TLS are not able to be mitigated exclusively through technical means. The corruption and resultant distrust of cryptographic standards is anathema to the security of TLS. Specifically, the unjustified methodologies used in the standardisation of named elliptic curves poses a real problem to the unconditional recommendation of forward secrecy. As we saw from analysis of popular websites, a worrying majority of services now prefer to use this functionality and very few do so with key moduli considered strong.

Background research showed that known cryptographic vulnerabilities were rarely practical to exploit, which is entirely contrary to the serious vulnerabilities repeatedly found in software

realisations. Additionally, it seems that optimising for performance through the introduction of complexity rarely comes without a cost to security: TLS compression introduced CRIME and code refactoring caused the Debian pseudo-random number generator weaknesses.

In conclusion, while I do not see application-level firewalling and network middleware as sustainable long-term solutions, I think TLSFilter is a pragmatic stopgap that will remain valuable until trusted, strong and securely-implemented cryptography becomes ubiquitous. Moreover, I think this project justifies the concept of decoupling the security of communication from application implementation instances and questioning the notion that application security should be enforced solely at the application level.

7.1 Future Extensions

TLS Tunnelling

The security-usability tradeoff is large drawback of TLSFilter. It would be useful to be able to securely multiplex and tunnel traffic between TLSFilter instances for connections that would otherwise be considered insecure and thus blocked. If deployment of TLSFilter were to hit a critical mass, a tightening of the default configuration to improve security would be possible with little impact to reachability.

Full TLS Implementation

At present TLSFilter does not attempt to implement the entire TLS specification so requests that contravene protocol standards to exploit underlying vulnerable implementations are not proactively detected and blocked. If TLSFilter were to be extended to support and perfectly enforce all protocol and state stipulations, this would reduce the extent to which bugs like Heartbleed can be exploited. However, this is not a trivial undertaking.

Greater Protocol Support

Greater protocol support would improve TLSFilter's ability to rapidly create plugins to mitigate new vulnerabilities. In particular, support for TLS "extensions", Datagram TLS (i.e., TLS over UDP) and the identification of SSL v2 traffic.

Add Parallel Processing

The single-threaded nature of TLSFilter is the likely cause of the 25% reduction in concurrent throughput (at 200 connections). In principle it should be possible to extend TLSFilter such that the execution of plugins is distributed over CPU cores or perhaps even to perform clustering of multiple TLSFilter instances.

Bibliography

[1] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R. and Polk, W. 2008. RFC 5280 - *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. [online] Available at: https://tools.ietf.org/html/rfc5280 [Accessed: 31 Jan 2014].

[2] Rescorla, E. and Dierks, T. 2008. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2*. [online] Available at: https://tools.ietf.org/html/rfc5246 [Accessed: 31 Jan 2014].

[3] Crypto++. n.d. *GCM Mode*. [online] Available at: http://www.cryptopp.com/wiki/GCM_Mode [Accessed: 31 Jan 2014].

[4] Dworkin, M. 2007. *The Galois/Counter Mode of Operation (GCM)*. [e-book] Gaithersburg: Available through: National Institute of Standards and Technology (NIST) http://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf [Accessed: 31 Jan 2014].

[5] Rescorla, E. and Dierks, T. 2006. *RFC 4346 - The Transport Layer Security (TLS) Protocol Version 1.1*. [online] Available at: https://tools.ietf.org/html/rfc4346 [Accessed: 31 Jan 2014].

[6] Rescorla, E. and Allen, C. 1999. *RFC 2246 - The TLS Protocol Version 1.0*. [online] Available at: https://tools.ietf.org/html/rfc2246 [Accessed: 31 Jan 2014].

[7] Freier, A., Karlton, P. and Kocher, P. 1996. *The SSL Protocol Version 3.0*. [online] Available at: https://tools.ietf.org/search/draft-ietf-tls-ssl-version3-00 [Accessed: 31 Jan 2014].

[8] Trustworthyinternet.org. 2014. Trustworthy Internet Movement - SSL Pulse. [online] Available at: https://www.trustworthyinternet.org/ssl-pulse/ [Accessed: 31 Jan 2014].

[9] Blake-Wilson, S., Bolyard, N., Gupta, B., Hakw, C. and Moeller, B. 2006. *RFC4492 - Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)*. [online] Available at: https://tools.ietf.org/html/rfc4492 [Accessed: 31 Jan 2014].

[10] Electronic Frontier Foundation. 2011. *The EFF SSL Observatory*. [online] Available at: https://www.eff.org/observatory [Accessed: 31 Jan 2014].

[11] Adkins, H. 2014. An update on attempted man-in-the-middle attacks. *Google Online Security Blog*, [blog] Monday, August 29, 2011 8:59 PM, Available at: http://googleonlinesecurity.blogspot.co.uk/2011/08/update-on-attempted-man-in-middle.html [Accessed: 31 Jan 2014].

[12] New Chromium security features, June 2011. 2011. *The Chromium Blog*, [blog] June 14, 2011, Available at: http://blog.chromium.org/2011/06/new-chromium-security-features-june.html [Accessed: 31 Jan 2014].

[13] Marlinspike, M. and Perrin, T. 2013. *Trust Assertions for Certificate Keys*. [online] Available at: https://tools.ietf.org/html/draft-perrin-tls-tack-02 [Accessed: 31 Jan 2014].

[14] Hoffman, P. and Schlyter, J. 2012. *RFC6698 - The DNS-Based Authentication of Named Entities (DANE)*. [online] Available at: https://tools.ietf.org/html/rfc6698 [Accessed: 31 Jan 2014].

[15] Lamb, R. 2014. DNSSEC Surpasses 50%!. *ICANN Blog*, [blog] 22 January 2014, Available at: http://blog.icann.org/2014/01/dnssec-surpasses-50/ [Accessed: 31 Jan 2014].

[16] Vaudenay, S. 2002. Security Flaws Induced by CBC PaddingApplications to SSL, IPSEC, WTLS... pp. 534–545.

[17] Alfardan, N. J., Bernstein, D. J., Paterson, K. G., Poettering, B. and Schuldt, J. 2013. On the security of RC4 in TLS and WPA.

[18] Ray, M. and Dispensa, S. 2009. Renegotiating TLS

[19] Rescorla, E., Ray, M., Dispensa, S. and Oskov, N. 2009. *Transport Layer Security (TLS) Renegotiation Indication Extension*. [online] Available at: https://tools.ietf.org/html/draft-rescorla-tls-renegotiation-00 [Accessed: 31 Jan 2014].

[20] Bard, G. V. 2006. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. pp. 99–109.

[21] BEAST. 2011. *thai*, [blog] 25 September 2011, Available at: http://vnhacker.blogspot.co.uk/2011/09/beast.html [Accessed: 31 Jan 2014].

[22] CRIME Ekoparty presentation slides. 2012. [video online] Available at: https://docs.google.com/presentation/d/11eBmGiHbYcHR9gL5nDyZChu_-lCa2GizeuOfaLU2HOU/edit [Accessed: 31 Jan 2014].

[23] Alfardan, N. J. and Paterson, K. G. 2013. Lucky thirteen: Breaking the TLS and DTLS record protocols.

[24] Davis, D., Ihaka, R. and Fenstermacher, P. 1994. Cryptographic randomness from air turbulence in disk drives. pp. 114–120.

[25] Debian.org. 2006. *Debian – Security Information – DSA-1571-1 openssl*. [online] Available at: http://www.debian.org/security/2008/dsa-1571 [Accessed: 31 Jan 2014].

[26] Goldberg, I. and Wagner, D. 1996. Randomness and the netscape browser. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, 21 (1), pp. 66–71.

[27] Backes, M., Doychev, G. and KÖpf, B. Preventing Side-Channel Leaks in Web Traffic: A Formal Approach.

[28] Netfilter.org. n.d. *Documentation about the Netfilter/iptables project*. [online] Available at: http://www.netfilter.org/documentation/ [Accessed: 31 Jan 2014].

[29] Mobius.illinois.edu. 2014. *The Mobius Tool*. [online] Available at: https://www.mobius.illinois.edu/ [Accessed: 31 Jan 2014].

[30] Ford, M. D., Keefe, K., Lemay, E., S, Ers, W. H. and Muehrcke, C. *Implementing the ADVISE Security Modeling Formalism in Mobius*.

[31] *Revealed: how US and UK spy agencies defeat internet privacy and security*. 2013. [online] 6th September. Available at: http://www.theguardian.com/world/2013/sep/05/nsa-gchq-encryption-codes-security [Accessed: 31 Jan 2014].

[32] *Schneier on Security: The NSA Is Breaking Most Encryption on the Internet*. 2013. [online] 5th September. Available at: https://www.schneier.com/blog/archives/2013/09/the_nsa_is_brea.html#c1675929 [Accessed: 28 May 2014].

[33] Chen, L. and Turner, S. 2011. *RFC 6151 - Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms*. [online] Available at: https://tools.ietf.org/html/rfc6151 [Accessed: 29 May 2014].

[34] Ptacek, T. and Newsham, T. (1998). *Insertion, evasion, and denial of service: Eluding network intrusion detection*.

[35] Adamic, L. and Huberman, B. (2001). *The Web's hidden order*. Communications of the ACM, [online] 44(9), pp.55-60. Available at: http://dl.acm.org/citation.cfm?doid=383694.383707 [Accessed 5 Jun. 2014].

[36] Adamic, L. and Huberman, B. (2002). *Zipfs law and the Internet*. Glottometrics, [online] 3(1), pp.143–150. Available at: http://www.hpl.hp.com/research/idl/papers/ranking/adamicglottometrics.pdf [Accessed 11 Jun. 2014].

[37] Schneier, B. (1999). *Attack trees*. Dr. Dobbs journal, 24(12), pp.21–29.

[38] Kleinjung, T., Aoki, K., Franke, J., Lenstra, A., Thomé, E., Bos, J., Gaudry, P., Kruppa, A., Montgomery, P., Osvik, D. and others, (2010). Factorization of a 768-bit RSA modulus. Springer, pp.333–350.

Appendix A

TLSFilter Configuration

A.1 Context-Free Grammar

$Configuration \rightarrow CommentedLine$ `EOL` $Configuration \mid \epsilon$

$CommentedLine \rightarrow Comment\ Line$
$Comment \rightarrow \#$ `String`

$Line \rightarrow GlobalSetting \mid PluginInvocation \mid NetworkScope \mid \epsilon$

$GlobalSetting \rightarrow$ `tlsfilter String`

$PluginInvocation \rightarrow PluginName\ Command$
$PluginName \rightarrow$ `String`
$Command \rightarrow$ `String` $Command \mid \epsilon$

$NetworkScope \rightarrow [Network, NetworkList]$
$NetworkList \rightarrow Network, NetworkList \mid \epsilon$

$Network \rightarrow IPAddress/SubnetMask \mid IPAddress$
$IPAddress \rightarrow Octet.Octet.Octet.Octet$

$Octet \rightarrow 0 \mid 1 \mid 2 \mid \ldots \mid 254 \mid 255$
$SubnetMask \rightarrow 1 \mid 2 \mid \ldots \mid 31 \mid 32$

Appendix B

Attack Tree Calculations

B.1 Notation

To make the calculations more clear, we can name the nodes in the attack tree from Figure 6.1 as shown in Figure B.1;

The value of the root is therefore = max(A,max(max(B,C,D),E,F×max(max(G,max(max(B,C,D), (max(J,K)×I),H)),L,max(B,C,D)),M,max(N,O,P),max(max(N,O,P),max(Q,R),S),T)).

TLSFilter can be configured to prevent the attacks corresponding to $A, C, E, G, H, K, L, O, Q, R, T, U, V, W$. So we can set these to zero in the previous form to produce: max(max(B,D),F×max(max (B,D)×J × I,max(B,D)), M,max(N,P),max(max(N,P), S)). Note that the parent of H has become an "AND" node, to take into account that a SHA-1 collision could be used to circumvent TLSFilter's certificate pinning.

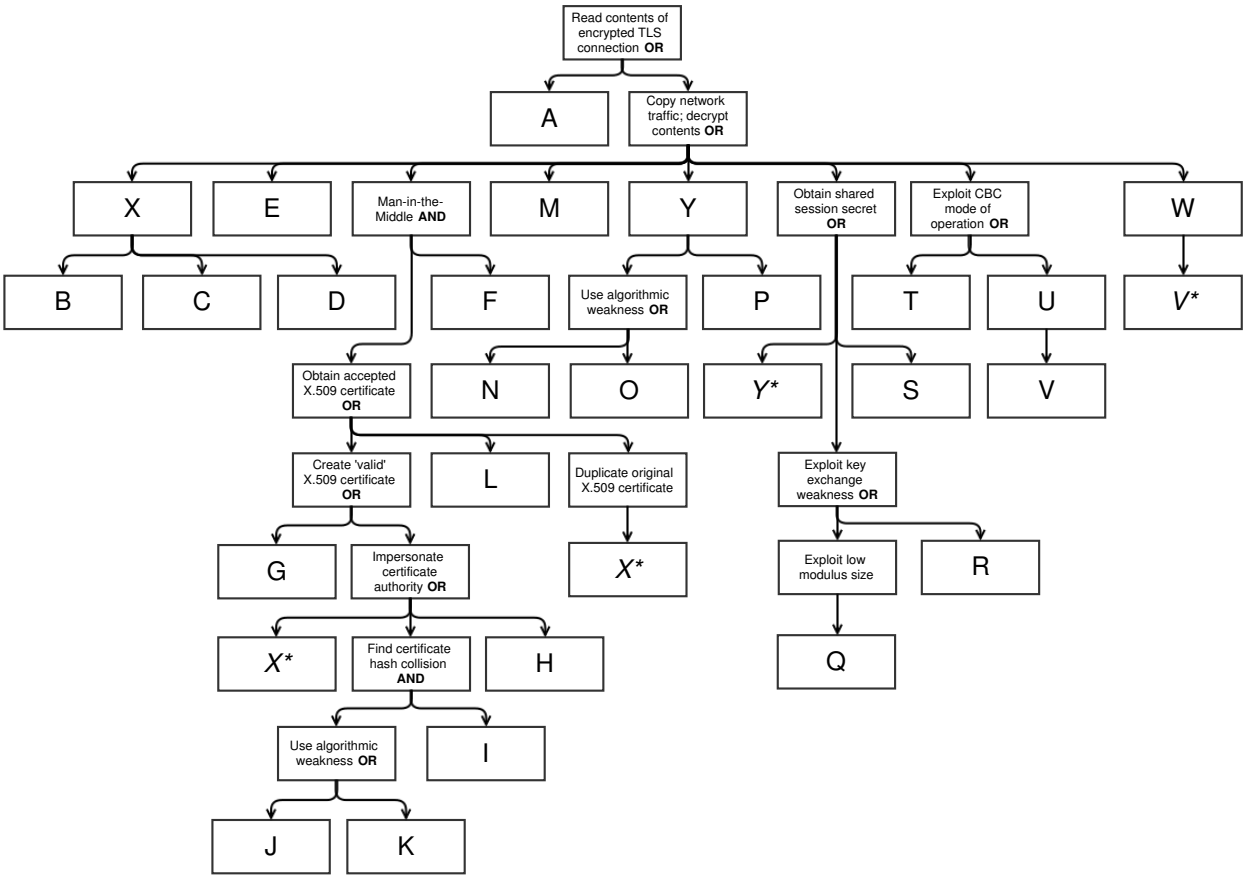


Figure B.1: Attack tree node key

B.2 Lavabit Estimated Probabilities

Scan data for `lavabit.com:443` obtained on Jun 5 2014 at 22:22:22 UTC. See Table B.1 for associated probabilities.

Node	Nation State	Criminal Gang	Lone Cracker	Comment
A	0.0	0.0	0.0	Not vulnerable to known bugs (e.g., Heartbleed)
B	0.8	0.0	0.0	Assumed friendly towards lawful intercept
C	0.8	0.4	0.2	Certificate chain contains a 1024-bit RSA key
D	0.2	0.1	0.1	
E	0.0	0.0	0.0	Not vulnerable to known bugs
F	1.0	0.2	0.0	
G	0.8	0.0	0.0	
H	0.2	0.2	0.0	
I	0.8	0.4	0.2	
J	0.2	0.0	0.0	
K	0.0	0.0	0.0	SHA-1 deprecated but not yet broken
L	1.0	1.0	1.0	Vulnerable to CCS MitM
M	0.0	0.0	0.0	No significant embedded content
N	0.2	0.0	0.0	3DES
O	0.0	0.0	0.0	No known practical attacks for AES, 3DES
P	0.0	0.0	0.0	AES and 3DES have large key spaces
Q	0.8	0.4	0.2	Weak 1024-bit DHE modulus used
R	0.0	0.0	0.0	ECC not used for key exchange
S	0.2	0.0	0.0	
T	0.0	0.0	0.0	Vulnerable, but impractical
U	0.0	0.0	0.0	Vulnerable, but impractical
V	0.0	0.0	0.0	
W	0.0	0.0	0.0	Not vulnerable

Table B.1: Node probability estimates for Lavabit, for each adversary

B.3 First Look Media Estimated Probabilities

Scan data for `firstlook.org:443` obtained on Jun 5 2014 at 15:50:48 UTC. See Table B.2 for associated probabilities.

Node	Nation State	Criminal Gang	Lone Cracker	Comment
A	0.0	0.0	0.0	Not vulnerable to known bugs (e.g., Heartbleed)
B	0.4	0.0	0.0	
C	0.2	0.0	0.0	All certificates are 2048-bit RSA or higher
D	0.2	0.1	0.1	
E	0.0	0.0	0.0	Not vulnerable to known bugs
F	1.0	0.2	0.0	
G	0.6	0.0	0.0	
H	0.2	0.2	0.0	
I	0.8	0.4	0.2	
J	0.2	0.0	0.0	
K	0.0	0.0	0.0	SHA-1 deprecated but not yet broken
L	1.0	1.0	1.0	Vulnerable to CCS MitM
M	0.0	0.0	0.0	Content served over SSL and of very similar sizes
N	0.2	0.0	0.0	
O	0.4	0.2	0.0	RC4 keystream biases
P	0.0	0.0	0.0	AES and RC4 where RC4 is preferred by most web browsers (not Internet Explorer)
Q	0.8	0.4	0.2	Weak 1024-bit DHE modulus used
R	0.8	0.0	0.0	
S	0.2	0.0	0.0	
T	0.0	0.0	0.0	Vulnerable, but impractical
U	0.0	0.0	0.0	Vulnerable, but impractical
V	0.0	0.0	0.0	
W	0.0	0.0	0.0	Not vulnerable

Table B.2: Node probability estimates for First Look Media, for each adversary

B.4 ESET Estimated Probabilities

Scan data for `secure.eset.co.uk:443` obtained on Jun 5 2014 at 21:53:58 UTC. See Table B.3 for associated probabilities.

Node	Nation State	Criminal Gang	Lone Cracker	Comment
A	0.0	0.0	0.0	Not vulnerable to known bugs (e.g., Heartbleed)
B	0.0	0.0	0.0	Based in Slovakia
C	0.2	0.0	0.0	All certificates are 2048-bit RSA or higher
D	0.2	0.1	0.1	
E	0.0	0.0	0.0	Not vulnerable to known bugs
F	1.0	0.2	0.0	
G	0.8	0.0	0.0	
H	0.2	0.2	0.0	
I	0.8	0.4	0.2	
J	0.2	0.0	0.0	
K	0.0	0.0	0.0	SHA-1 deprecated but not yet broken
L	0.0	0.0	0.0	Not vulnerable to known bugs
M	0.0	0.0	0.0	
N	0.6	0.0	0.0	In reference to RC4
O	1.0	1.0	1.0	DES, RC2 and RC4 all supported
P	1.0	0.8	0.6	Supports export grade 40-bit ciphers
Q	1.0	0.8	0.6	No forward secrecy; relies on symmetric cipher strength
R	0.0	0.0	0.0	ECC not used for key exchange
S	0.2	0.0	0.0	
T	0.0	0.0	0.0	Not vulnerable
U	0.0	0.0	0.0	Vulnerable, but impractical
V	0.0	0.0	0.0	
W	0.0	0.0	0.0	Not vulnerable

Table B.3: Node probability estimates for ESET, for each adversary