To Intercept or Not to Intercept: Analyzing TLS Interception in Network Appliances

Louis Waked, Mohammad Mannan, and Amr Youssef
Concordia Institute for Information Systems Engineering
Concordia University
Montreal, Canada
{l_waked,mmannan,youssef}@ciise.concordia.ca

ABSTRACT

Many enterprise-grade network appliances host a TLS proxy to facilitate interception of TLS-protected traffic for various purposes, including malware scanning, phishing detection, and preventing data exfiltration. When deployed, the TLS proxy acts as the security validating client for external TLS web servers, on behalf of the original requesting client; on the other hand, the proxy acts as the web server to the client. Consequently, TLS proxies must maintain a reliable level of security, at least, at the same level as modern web browsers and properly configured TLS servers. Failure to do so increases the attack surface of all the proxied clients served the network appliance. We develop a framework for testing TLS inspecting appliances, combining and extending tests from existing work on client-end and network-based interception. Utilizing this framework, we analyze six representative network appliances, and uncover several security issues regarding TLS version and certificate parameters mapping, CA trusted stores, private keys, and certificate validation tests. For instance, we found that two appliances perform no certificate validation at all, exposing their end-clients to trivial Man-in-the-Middle attacks. The remaining appliances that perform certificate validation, still do not follow current best practices, and thus making them vulnerable against certain attacks. We also found that all the tested appliances deceive the requesting clients, by offering TLS parameters that are different from the proxy-to-server TLS parameters, such as the TLS versions, hashing algorithms, and RSA key sizes. We hope that this work bring focus on the risks and vulnerabilities of using TLS proxies that are being widely deployed in many enterprise and government environments, potentially affecting all their users and systems.

CCS CONCEPTS

• **Networks → Middle boxes / network appliances**; • **Security and privacy → Network security**; Browser security;

KEYWORDS

TLS, Proxy, Interception, Network Appliances, Certificates, Validation, MITM

ACM Reference Format:
Louis Waked, Mohammad Mannan, and Amr Youssef. 2018. To Intercept or Not to Intercept: Analyzing TLS Interception in Network Appliances. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3196494.3196528

1 INTRODUCTION

Most network appliances currently include an SSL/TLS interception feature in their products. The interception process is performed by making use of a TLS web proxy server. Being either transparent or explicit to the end-user, the proxy intercepts the user's request to visit a TLS server, and creates two separate TLS connections. It acts as the HTTPS endpoint for the user's browser, and as the client for the actual HTTPS web server. Having the appropriate private key for the signing certificate (inserted to the client's root CA store), the proxy has access to the raw plaintext traffic, and can perform any desired action, such as restricting the access to the web page by parsing its content, or passing it to an anti-virus/malware analysis module or a customized traffic monitoring tool. Common reasons for adopting TLS interception include the protection of organization and individuals against malware and phishing attacks, law enforcement and surveillance, access control and web filtering, national security, hacking and spying, and privacy and identity theft [44].

While interception violates the implicit end-to-end guarantee of TLS, we focus on the potential vulnerabilities that such feature introduces to end-users located behind the network appliances, following several other existing studies on TLS interception, e.g., [6, 17, 32, 35, 41]. In general, TLS interception, even if implemented correctly, still increases the attack surface on TLS due to the introduction of an additional TLS client and server at the proxy. However, the lack of consideration for following the current best practices on TLS security as implemented in modern browsers and TLS servers, may result in severe potential vulnerabilities, and overall, a significantly weak TLS connection.

For example, the proxy may not mirror the TLS version and certificate parameters or might accept outdated, insecure ones. Also, the proxy could allow TLS compression, enabling the CRIME attack [34], or insecure renegotiation [43]. The proxy may downgrade the Extended Validation (EV) domains to Domain Validated (DV) ones. The proxy also may not mirror the cipher suites offered by the requesting client, and use a hard-coded list with weak and insecure ciphers, reviving old attacks such as FREAK [27], Logjam [26], and BEAST [33]. If the proxy does not implement a proper certificate validation mechanism, invalid and tampered certificates could be accepted by the proxy, and the clients (as they see only proxy-issued, valid certificates). Accepting its own root certificate as the signing authority of externally delivered content could allow MITM attacks on the network appliance itself. The use of a pre-generated key pair by a proxy could enable a generic trivial MITM attack [32]. In addition, the proxy may rely on an outdated root CA store for certificate validation, containing certificates with insecure key length, expired certificates, or banned certificates that are no longer trusted by major browsers/OS vendors.

Concerns about security weaknesses introduced by TLS interception proxies are not new. In 2012, Jarmoc [41] proposed a basic framework for testing network appliances consisting of seven certificate validation tests, and applied it on four network appliances. Dormann [6, 17] relied on badssl.com's tests to analyze the certificate validation process of two network appliances, revealing flaws in the appliances' certificate validation mechanisms. Carnavalet and Mannan [32] designed a framework for analyzing client-based TLS proxies (as included in several leading anti-virus and parental control applications), and revealed several flaws in the TLS version and certificate mapping, certificate validation, private key generation and protection, CA trusted store content, in addition to vulnerabilities to known TLS attacks. In 2017, Durumeric et al. [35] applied tests from earlier frameworks on 12 network appliances and 13 client-side TLS proxies, uncovering several flaws in certificate validation, cipher suites, TLS versions and known TLS attacks.

We argue that most past studies on network appliances analyzed only preliminary aspects of TLS interception, while the extensive work of Carnavalet and Mannan [32] targeted only client-end TLS proxies. However, TLS vulnerabilities in network appliances could result in more serious security issues, as arguably, enterprise computers handle more important business/government data in bulk, compared to personal information on a home user machine. Also, a single, flawed enterprise TLS proxy can affect hundreds of business users, as opposed to one or few users using a home computer with a client-side TLS proxy.

In this work, we present an extensive framework dedicated for analyzing TLS intercepting appliances, borrowing/adapting several aspects of existing work on network appliances and client-end proxies, in addition to applying a set of comprehensive certificate validation tests. We analyze the TLS-related behaviors of appliance-based proxies, and their potential vulnerabilities from several perspectives: TLS version and certificate parameter mapping, cipher suites, private key generation/protection, content of root CA store, known TLS attacks, and 32 certificate validation tests. We use this framework to evaluate six representative TLS network appliances between July and October 2017 (see Table 1), including open source, free, low-end, and high-end network appliances, and present the vulnerabilities and flaws found. All our findings have been disclosed to the respective companies.

We found that two network appliances do not perform *any* certificate validation, enabling simple MITM attacks against their clients. One appliance performs TLS certificate caching, also leading to trivial MITM attacks. Four appliances accept their own certificates for externally delivered content. Four appliances offer weak and insecure ciphers, while one appliance only accepts TLS 1.0 and SSL 3.0. We also found that the root CA stores of all appliances include certificates deemed untrusted by major browser/OS vendors, and one appliance includes an RSA-512 certificate, which can be trivially compromised. Five appliances also do not encrypt their private keys; three such keys are accessible by unprivileged processes running on the same appliance.

Table 1: List of the tested appliances

Appliance	Company	Version
Untangle NG Firewall	Untangle	13.0
pfSense	NetGate	2.3.4
WebTitan Gateway	TitanHQ	5.15 build 794
Microsoft TMG	Microsoft	2010 (SP1, SP1 Update, SP2 rollup updates 1 to 5)
UserGate Web Filter	Entensys	4.4.3320601
Cisco Ironport WSA	Cisco	Async OS version 10.5.1 build 270

Analyzing network appliances raises several new challenges compared to testing browsers and client-end TLS proxies. Several network appliances do not include an interface for importing custom certificates (essential for testing), and many appliances do not provide access to the file system or a terminal, overburdening the tasks of injecting custom certificates and locating the signing keys (for details, see Appendix B). Many appliances do not support more than one or two network interfaces, and thus, require the use of a router to connect to multiple interfaces. In addition, appliances that perform SSL certificate caching require the generation of a new root key pair for their TLS proxies for each test.

Our contributions can be summarized as follows: (1) We develop a comprehensive framework to analyze TLS interception in enterprise-grade network appliances, combining our own certificate validation tests with existing tests for TLS proxies (both client-end services and network appliances), which we reuse or adapt as necessary for our purpose. Our certificate validation tests can be found at: https://madiba.encs.concordia.ca/software/tls-netapp/. (2) We use this framework to evaluate six well-known appliances from all tiers: open source, free, low-end, and high-end products, indicating that the proposed framework can be applied to different types of network appliances. (3) We uncover several vulnerabilities and bad practices in the analyzed appliances, including: either an incomplete or completely absent certificate validation process (resulting trivial MITM attacks), improper use of TLS parameters that mislead clients, inadequate private key protection, and the use of weak/insecure cipher suites.

2 BACKGROUND

In this section, we describe the TLS interception process, list the tested products, state the expected behavior of a TLS proxy, and explain the threat model.

Terminology. Throughout the paper, we refer to the TLS intercepting network appliances as proxies, HTTPS proxies, TLS proxies, middleboxes, or simply appliances. For the TLS requesting client, we use: browser, end-user, user, or client. The term *mirroring* is used to describe a situation where the proxy sends the same TLS parameters received from the web server to the client side, and vice versa; otherwise, *mapping* is used to indicate that the proxy has modified some parameters (for better or worse). Finally, we refer to virtual machines as virtual appliances, VMs, or simply machines.

2.1 Proxies and TLS Interception

For TLS interception, network appliances make use of TLS proxies, deployed as either transparent proxies or explicit proxies. The explicit proxy requires the client machine or browser to have the proxy's IP address and listening port specifically configured to operate. Thus, the client is aware of the interception process, as the requests are sent to the proxy's socket. On the other hand, transparent proxies may operate without the explicit awareness of the clients, as they intercept outgoing requests that are meant for the web servers, without the use of an explicit proxy configuration on the client side; however, for TLS interception, a proxy's certificate must be added to the client's trusted root CA store (explicitly by the end-user, or pre-configured by an administrator). Such proxies could filter all ports, or a specific set of ports, typically including HTTP port 80 and HTTPS port 443. Secure email protocols could also be intercepted, by filtering port 465 for secure SMTP, port 993 for secure IMAP, and port 995 for secure POP3. The proxy handles the client's outgoing request by acting as the TLS connection's endpoint, and simultaneously initiates a new TLS connection to the actual web server by acting as the client, while relaying the two connections' requests and responses.

By design, the TLS protocol should prevent any MITM interception attempt, by enforcing a certificate validation process, which mandates that the incoming server certificate must be signed by a trusted issuer. Certificate authorities only provide server certificates to validated domains, and not to forwarding proxies, precluding the proxy from becoming a trusted issuer (i.e., a *valid* local CA). To bypass this restriction, the proxy can use a self-signed certificate that is added to the trusted root CA store of the TLS client, and thereby allowing the proxy to sign certificates for *any* domain on-the-fly, and avoid triggering browser warnings that may expose the untrusted status of the proxy's certificate. Thereafter, all HTTPS pages at the client will be protected by the proxy's certificate, instead of the intended external web server's certificate. Users are not usually aware of the interception process, unless they manually check the server certificate's issuer chain, and notice that the issuer is a local CA [42].

2.2 Tested Appliances

Most current network appliance vendors offer products for TLS interception. We select six products, including: free appliances, appliances typically deployed by small companies, appliances with affordable licensing for small to medium sized businesses, and high-end products for large enterprises; see Table 1. On a side note, we performed several rounds of updates and patches for Microsoft Threat Management Gateway, on a Windows Server 2008 R2 operating system, as recommended by Microsoft's documentation [13]. These include the service pack 1 (SP1), the service pack 1 update, the service pack 2, and five rollup updates (1 to 5) [12].

For all the analyzed appliances, we keep the default configuration for their respective TLS proxies. An administrator could of course manually modify this default configuration, which may improve or damage the proxy's TLS security. We thus choose to apply our test framework on the unmodified configuration (assuming the vendors will use *secure-defaults*).

2.3 Expected Behavior of a TLS Proxy

We summarize TLS proxy behaviors as expected from a prudent interception proxy (following [32]). Deviations from these behaviors help design and refine our framework and validation tests.

The TLS version, key length, and signature algorithms should be mirrored (between client-proxy and proxy-web) to avoid misleading clients regarding the TLS security parameters used in the proxy to external web server connection. The list of cipher suites offered by the client should ideally be mirrored to the server's TLS connection, or at least maintained to have no weak/insecure ciphers. Domains with EV certificates should not be downgraded to DV certificates, by exempting them from the interception process (e.g., through white-listing, or simply based on the certificate type). The TLS proxies and any associated libraries (e.g., OpenSSL, GnuTLS) must be up-to-date, and patched against known TLS attacks and vulnerabilities (following major browser vendors), such as BEAST [33], CRIME [34], FREAK [27], Logjam [26], and TLS insecure renegotiation [43].

Typically, the client software (e.g., a web browser) is the last line of defense against faulty external certificates, as it is the sole entity responsible for the received certificate's chain of trust validation. When deployed, the TLS proxy takes on the responsibility of protecting the clients by performing a proper certificate validation on behalf of them, as the browser will be only exposed to the proxy-issued certificates. A less-stringent or incomplete certificate validation process could result in severe consequences, e.g., enabling MITM and downgrading attacks on client-based TLS proxies [32]. The impact is even higher when a network appliance's TLS proxy lacks strict TLS validation, affecting many enterprise machines behind the appliance. Thus, all aspects of TLS chain of trust should be properly validated, checking for flaws such as untrusted issuers, mismatched signatures, wrong common-names, constrained issuers, revoked and expired certificates, certificate usage, short key certificates and deprecated signature algorithms. TLS proxies should also recognize their own root certificate if provided by an external web server (which should never happen), and block such connections. Also, the proxy's trusted CA store must not include short key, expired or untrusted issuer certificates.

Vendors should adequately protect proxies' private keys, e.g., by encrypting them, and limiting access permissions to the root account. The keys must not be pre-generated, to limit the aftermath of a leaked private key from a single product, avoiding cases such as Lenovo's SuperFish [11].

2.4 Threat Model

We mainly consider three types of attackers.

An *external attacker* can impersonate any web server by performing a MITM attack on a network appliance that does not perform a proper certificate validation. The attacker could be anywhere on the network between the appliance and the target website. Even if the validation process is perfect, the attacker could still impersonate any web server, if the appliance uses a pre-generated root certificate and accepts external site-certificates signed by its own root key. The attacker could also take advantage of known TLS attacks/vulnerabilities to potentially acquire authentication cookies (BEAST, CRIME), or impersonate web servers (FREAK, Logjam).

A *local attacker* (e.g., a malicious employee) with a network sniffer in promiscuous mode can get access to the raw traffic from the connections between the network appliance and clients. If the appliance uses a pre-generated certificate, the malicious user can install his own instance of the appliance, acquire its private key, and use it to decrypt the sniffed local traffic when the TLS connections are not protected by forward-secure ciphers. Such an adversary can also impersonate the proxy itself to other client machines, although this may be easily discovered by network administrators.

An *attacker who compromises the network appliance* itself with non-root privileges can acquire the private key if the key is not properly protected (e.g., read access to 'other' users and no passphrase encryption). With elevated privileges, more powerful attacks can be performed (e.g., beyond accessing/modifying TLS traffic). We do not consider such privileged attackers, assuming having root access on the appliance would be much more difficult than compromising other low-privileged accounts. Note that, in most cases, the *appliance* is simply an ordinary Linux/Windows box with specialized software/kernel, resulting a large trusted code base (TCB).

3 RELATED WORK

Several studies have been recently conducted on TLS interception, TLS certificate validation, and forged TLS certificates. We briefly review studies that are closely related to our work.

Interception. Jarmoc [41] uncovered several TLS vulnerabilities in the certificate validation process of four network appliances using a test framework with seven certificate validation checks. Dormann [6, 17] relied on badssl.com's tests to check for vulnerabilities in two network appliances, finding flaws in the certificate validation process and the acceptance of insecure TLS parameters. Dormann also compiled a list of possibly affected software and hardware appliances.

Carnavalet and Mannan [32] proposed an extensive framework for analyzing client-end TLS intercepting applications, such as antivirus and parental control software. They analyzed 14 applications (under Windows 7), revealing major flaws such as pre-generated certificates, faulty certificate validation, insecure private key protection, improper TLS parameter mapping, vulnerabilities to known TLS attacks, and unsanitized trusted CA stores. Durumeric et al. [35] later additionally included 5 TLS proxies under Mac OS, and 12 network appliances. They found that TLS proxies under Mac OS introduce more flaws than their Windows counterparts. They also showed that web servers can detect TLS interception, through the HTTP User-Agent header and protocol fingerprinting.

In March 2017, US-CERT [22] published an alert regarding TLS interception, to raise awareness of the dangers of TLS interception and its impact. Ruoti et al. [44] surveyed 1976 individuals regarding TLS inspection, to understand user opinion regarding legitimate uses of TLS inspection. Over 60% of the surveyed individuals had a negative response towards TLS inspection, and cited malicious hackers and governments as their main concerns.

Certificates scans. Huang et al. [40] analyzed over three million real-world TLS connections to facebook.com to detect forged certificates. They found that around 0.2% of the analyzed connections make use of a forged certificate, caused mainly by anti-virus software, network appliances and malware. O'Neill et al. [42] analyzed over 15 million real-world TLS connections using Google AdWords campaigns. They found that nearly 0.4% of the TLS connections were intercepted by TLS proxies, mostly by anti-virus products and network appliances, with the highest interception rates in France and Romania. In addition, Issuer Organization fields in some certificates matched the names of malware, such as 'Sendori, Inc', 'Web-MakerPlus Ltd', and 'IopFailZeroAccessCreate'.

Certificates validation. Fahl et al. [36] analyzed 13,500 free Android apps for MITM vulnerabilities. They found that 8% of the analyzed apps contain potentially vulnerable TLS modules. They also performed manual inspection of 100 apps, and successfully executed MITM attacks on 41, capturing credentials for widely used commercial and social websites, e.g., Google, Facebook, Twitter, Paypal, and several banks. Their attacks relied on exploiting flaws in the certificate validation process; many apps ignored the chain of trust validation, accepting self-signed certificates, and mismatched common names.

Georgiev et al. [37] demonstrated that several widely used applications and development libraries, such as Amazon's EC2 Java library, Amazon and Paypal's SDK, osCommerce, and Java web services, among others, suffered from certificate validation vulnerabilities, leading to generic MITM attacks. These vulnerabilities were attributed to be caused by (primarily) the use of poorly designed APIs, such as JSSE and OpenSSL.

Brubaker et al. [30] designed an automated approach for testing the certificate validation modules of several well-known TLS implementations. They first scanned the Internet for servers with port 443 open using ZMap [25], and collected all the available certificates. Then, they permuted the certificate parameters and possible X509 values, compiling a list of 8 million *Frankencerts*. Using Frankencerts and differential testing, Brubaker et al. found over 200 discrepancies in these commonly used TLS implementations (e.g., OpenSSL, GnuTLS and NSS). He et al. [38] designed an automated static analysis tool for analyzing TLS libraries and applications. They then evaluated Ubuntu 12.04 TLS packages, and found 27 zero-day TLS vulnerabilities, related to faulty certificate/hostname validation.

Sivakorn et al. [45] proposed a black-box hostname verification testing framework for TLS libraries and applications. They evaluated the hostnames accepted by seven TLS libraries and applications, and found eight violations, including: invalid hostname characters, incorrect null characters parsing, and incorrect wildcard parsing.

Chau et al. [31] made use of a symbolic execution approach to test the certificate validation process of nine TLS libraries, compared to RFC 5280 [39]. They found 48 instances of noncompliance; libraries ignored several X509 certificate parameters, such as the pathLenConstraint, keyUsage, extKeyUsage, and 'notBefore' validity dates.

Comparison. The most closely related work is by Durumeric et al. [35] (other studies mostly involved analyzing TLS libraries and

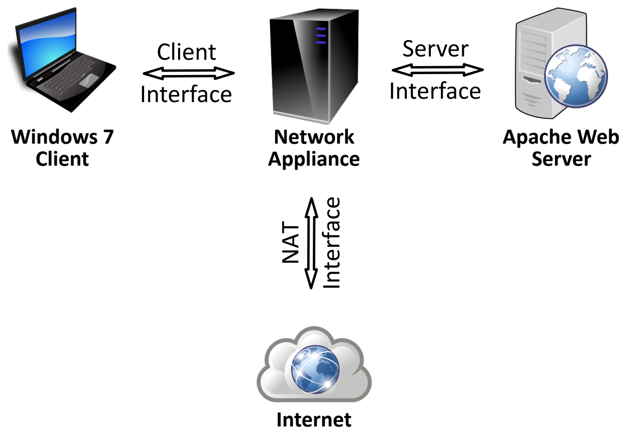

Figure 1: Framework components and the overall test architecture

client-end proxies). While their work focuses primarily on fingerprinting TLS interception, in addition to a brief security measurement for several HTTPS proxies, we develop an extensive framework dedicated for analyzing the TLS interception on network appliances. They checked/rated the highest TLS version supported by a target proxy, while we examine all the supported versions by the proxy, in addition to their respective mapping/mirroring to the client side. Durumeric et al.'s certificate validation tests include: expired, self-signed, invalidly signed certificates, and certificates signed by CAs with known private keys; we include more tests for this important aspect (a total of 32 distinct tests). We also include several new tests such as: checking the content of the CA trusted store and the certificate parameter mapping, locating the private signing keys of the proxies and examining their security (including checking pre-generated root certificates); these tests are mostly added/extended from [31, 32].

In terms of results, for the four overlapping products with Durumeric et al. [35], we observed a few differences; note that our analysis was performed on newer versions of the products and Durumeric et al. shared their results to the affected vendors more than a year prior to our tests. They found that WebTitan Gateway had a broken certificate validation process and offered RC4 and modern ciphers; we found that WebTitan does not perform *any* certificate validation and still offers RC4, in addition to weak ciphers, with 3DES and IDEA. We found that Untangle no longer offers RC4 ciphers (3DES is offered). Cisco Ironport WSA no longer offers RC4 and export-grade ciphers. According to [35], Microsoft TMG performed no certificate validation and the highest supported SSL/TLS version was SSLv2.0; it now performs certificate validation, and supports SSL versions 2.0, 3.0 and TLS 1.0.

4 PROPOSED FRAMEWORK

In this section, we present the setup/architecture of the proposed framework, and the major components and tests included in it.

4.1 Test Setup/Architecture

Our framework consists of three virtual machines: a client, a web server, and the TLS intercepting network appliance; see Figure 1. The client machine (Windows 7 SP1) is located behind the network

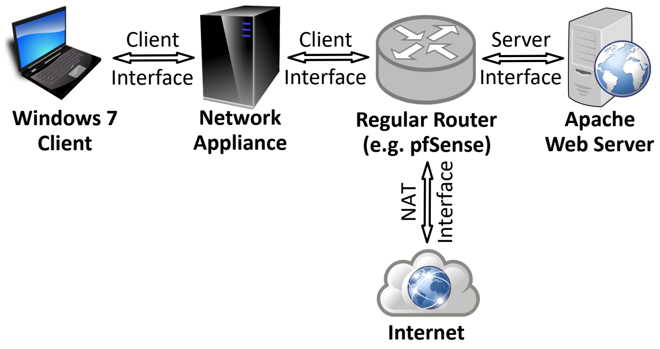

Figure 2: Framework components and test architecture with a router

appliance; we update the client with all available Windows updates, and install up-to-date Mozilla Firefox, Google Chrome, and Internet Explorer 11 on it. We insert the TLS proxy's root certificate into the client's trusted root stores (both Windows and Mozilla stores). We use a browser to initiate HTTPS requests to our local Apache web server, and the online TLS security testing suites (for certain tests). These requests are intercepted by the TLS proxy being analyzed.

The second machine hosts a web server (Apache under Ubuntu 16.04), configured to accept HTTP requests on port 80 and HTTPS requests on port 443; all port 80 requests are redirected to port 443. The web server is initially configured to accept all TLS/SSL protocol versions, and all available cipher suites. The server name is configured to be *apache.host*, as the crafted certificates must hold a domain name instead of an IP address. We generate the faulty certificates using OpenSSL, which are served from the Apache web server. It also hosts the patched howsmyssl.com code [10].

The pre-installed OpenSSL version on the Ubuntu 16.04 distribution is not compiled with SSLv3 support. Thus, in order to test the acceptance and mapping of SSLv3 only, we rely on an identically configured older version of Ubuntu (14.04), with an older OpenSSL version that supports SSLv3.

The third machine is the virtual network appliance that we want to test. These appliances are typically available as a trial version from a vendor's website, with a pre-configured OS, either as an ISO image file or an Open Virtualization Format file. The appliances are configured to intercept TLS traffic either as a transparent or explicit proxy, depending on the available modules. If both are available, transparent proxies are prioritized, as they do not require any client-side network configuration. We disable services such as firewall and URL filtering, if bundled in the appliances, to avoid any potential interferences in our TLS analysis. The root CA certificates corresponding to our faulty test certificates are injected into the trusted stores of the network appliances.

We setup a local DNS entry for *apache.host* on the client, web servers and network appliances machines. Operating systems match local DNS entries, found typically in the *hosts* file, before remote DNS entries, resulting in the correct mapping of our test server's domain name to its corresponding IP address.

The framework requires the use of three different interfaces on each virtual network appliance. The *Client Interface* is used to connect to the Windows 7 client. The traffic incoming from this interface is intercepted by the TLS proxy. Transparent proxies only require the network appliance to be the default gateway for the client, while explicit proxies require the client to configure the proxy settings with the appliance's socket details. The *Server Interface* is used to connect to the Apache web server. The *WAN Interface* is used to connect to the Internet, through Network Address Translation (NAT). However, some appliances support one or two interfaces. In such cases, we add a fourth virtual machine, that acts solely as a router with multiple interfaces. We use pfSense as the router (without TLS interception), relying on it for NATting and routing traffic of the three interfaces required in our setup; see Figure 2. The client and the network appliance are connected to the *Client Interface*, the web server is connected to the *Server Interface*, and the Internet connectivity is provided through NAT on a third interface on via pfSense. A local DNS entry for *apache.host* is also added to this router.

4.2 CA Trusted Store

We first need to locate the CA trusted store of the TLS proxy. This allows us to inject our root CA certificates into the stores, required for most of our tests.

Injecting custom certificates into a trusted store could be trivial, if the appliance directly allows adding custom root CAs (e.g., via its user interface). If no such interface is offered, we attempt to get a command line (*shell*) access through the SSH service, if available, by enabling the SSH server first through the settings panels. If SSH is unavailable, we mount the virtual disk image of the appliance on a separate Linux machine. When mounting, we perform several attempts to find the correct filesystem type and subtype used by the appliance (undocumented). After a successful mount, we search the entire filesystem for digital certificates in known formats, such as ".crt", ".pem", ".cer", ".der", and ".key". We thus locate several directories with candidate certificates, and subsequently delete the content of each file, while trying to access regular websites from the client. When an "untrusted issuer" warning appears at the client, we then learn the exact location/directory of the trusted store. This methodology is used to eliminate duplicate certificates found in several directories.

We then inject the custom crafted root certificates into the trusted CA stores. We also parse (via custom scripts) the certificates available in the trusted stores to identify any expired certificates, or certificates with short key lengths (e.g., RSA-512 and RSA-1024). We also check for the presence of root CA certificates from issuers that are no longer trusted by major browser/OS vendors. Our list of misbehaving CAs includes: China Internet Network Information Center (CNNIC [4]), TÜRKTRUST [19], ANSSI [16], woSign [5], Smartcom [5], and Diginotar [3].

4.3 TLS Version Mapping

To test the SSL/TLS version acceptance and TLS parameter mapping/mirroring, we alter the Apache web server's configuration. We use a valid certificate whose root CA certificate is imported into the trusted stores of the client (to avoid warnings and errors). We then subsequently force one TLS version after another at the web server, and visit the web server from the client, while documenting the versions observed in the browser's HTTPS connection information. Using this methodology, we are able to analyze the behavior of a proxy regarding each SSL/TLS version: if a given version is blocked, allowed, or altered in the client-to-proxy HTTPS connection.

4.4 Certificate Parameters Mapping

We check if the proxy-to-server certificate parameters are mapped or mirrored to the client-to-proxy certificate parameters. The parameters studied are signature hashing algorithms, certificate key lengths, and the EV/DV status.

For testing signature hashing algorithms, we craft multiple valid certificates with different *secure* hash algorithms, such as SHA-256, SHA-384 and SHA-512. We import their root CA certificates into the trusted stores of the client to avoid warnings and errors. We subsequently load each certificate and its private key into the web server, and visit the web page from the browser. We track the signature algorithms used in the certificates generated by the TLS proxy for each connection, and learn if the proxy mirrors the signature hashing algorithms, or use a single hard-coded one.

For testing certificate key lengths, we craft multiple certificates with different acceptable key sizes, such as RSA-2048, RSA-3072 and RSA-4096. We import their correspondent root CA certificates into the trusted stores of the client. We subsequently load each certificate and its private key into the web server, and visit the web page from the browser. We check the key length used for the client-to-proxy server certificate generated by the TLS proxy for each connection, and learn if the proxy mirrors the key-length, or uses on a single hard-coded length.

We rely on Twitter's website to study the network appliance's behavior regarding EV certificates. We visit twitter.com on the client machine, and check the client-to-proxy certificate displayed by the browser. TLS proxies can identify the presence of EV certificates (e.g., to avoid downgrading them to DV), by parsing the content and locating the CA/browser forum's EV OID: 2.23.140.1.1 [7].

4.5 Cipher Suites

Cipher suites offered by the TLS proxy in the proxy-to-server TLS connection can be examined in multiple ways. We initially rely on publicly hosted TLS testing suites, howsmyssl.com and the Qualys client test [18]. Since the connection is proxied, the displayed results found on the client's browser are the results of the proxy-to-server connection, and not the client-to-proxy connection. If the mentioned web pages are not filtered, for reasons such as the use unfiltered or non-standard ports, we use Wireshark to capture the TLS packets and inspect the Client Hello message initiated by the proxy to locate the list of ciphers offered.

We then compare the list of ciphers offered by the proxy to the list of ciphers offered by the browsers, learning if the TLS proxy performs a cipher suite mirroring or uses a hard-coded list. We also parse the list of ciphers offered by the proxy for weak and insecure ciphers that could lead to insecure and vulnerable TLS connections.

4.6 Known TLS Attacks

We test TLS proxies for vulnerabilities against well-known TLS attacks, including: BEAST, CRIME, FREAK, Logjam, and Insecure Renegotiation. We rely on the Qualys SSL Client Test [18] to confirm if the TLS proxy is patched against FREAK, Logjam, and Insecure

Renegotiation, in addition to checking if TLS compression is enabled, which could result in a possible CRIME attack. We visit the web page from the client browser, which displays the results for the proxy-to-server TLS connection. Regarding the BEAST attack, we rely on howsmyssl.com [10], with the modifications from Carnavalet and Mannan's [32] to test the proxies that support TLS 1.2 and 1.1.

4.7 Crafting Faulty Certificates

We use OpenSSL to craft our invalid test certificates, specifying *apache.host* as the Common Name (CN), except for the wrong CN test. We then deploy each certificate on our Apache web server, and request the HTTPS web page from the proxied client, and thus learn how the TLS proxy behaves when exposed to faulty certificates; if a connection is allowed, we consider the proxy is at fault. If the proxy replaces the faulty certificate with a valid one (generated by itself), leaving no way even for a prudent client (e.g., an up-to-date browser) to detect the faulty remote certificate, we consider this as a serious vulnerability. If the proxy passes the unmodified certificate and relies on client applications to react appropriately (e.g., showing warning/error messages, or terminating the connection), we still consider the proxy to be at fault for two reasons: (a) we do not see any justification for allowing plain, invalid certificates by any TLS agent, and (b) not all TLS client applications are as up-to-date as modern browsers, and thus may fail to detect the faulty certificates.

When the certificate's chain of trust contain intermediate certificate(s), we place the leaf certificate and intermediate certificate(s) at the web server, by appending the intermediate certificate(s) public keys after the server leaf certificate, in *SSLCertificateFile*. Note that we inject the issuing CA certificates of the crafted certificates into the TLS proxy's trusted store for all tests, except for the unknown issuer test and the fake GeoTrust test.

We enumerate the list of invalid certificate validation tests that we used (for details, see Appendix A); we compile this list using several sources (including [31, 32, 39]).

- Self-signed Certificate: A leaf certificate whose issuer is itself.
- Signature Mismatch: A leaf certificate with a tempered signature.
- Fake GeoTrust: A leaf certificate without an Authority Key Identifier, and whose untrusted issuer has the same subject name as the GeoTrust root CA.
- Wrong CN: A leaf certificate with a CN not matching *apache.host*.
- Unknown Issuer: A leaf certificate with an untrusted issuer.
- Non-CA Intermediate: An intermediate certificate with the CA basic constraint parameter set to be false.
- X509v1 Intermediate: An intermediate X509v1 certificate with no CA basic constraint parameter.
- Invalid pathLenConstraint: An intermediate certificate with a pathLenConstraint of 0 issuing another intermediate certificate.
- Bad Name Constraint Intermediate: An intermediate certificate constrained for a different domain issues a leaf certificate for *apache.host*.
- Unknown Critical X509v3 Extension: A leaf certificate with an unknown certificate extension object identifier (OID), set to critical.
- Malformed Extension Values: A leaf certificate with an atypical value for a certificate extension.
- Revoked: A leaf certificate issued by a revoked issuer.
- Expired Leaf, Intermediate and Root: Three tests with either an expired leaf, intermediate or root certificate.
- Not Yet Valid Leaf, Intermediate and Root: Three tests with either a leaf, intermediate or root certificate, which is not yet valid.
- Wrong keyUsage in Leaf and Root: Two tests with invalid keyUsage parameters for a root and a leaf certificate.
- Wrong extKeyUsage in Leaf and Root: Two tests with invalid extKeyUsage parameters for a root and a leaf certificate.
- Short Key Length in Root and Leaf: Multiple tests using short key lengths for root and leaf certificates.
- Bad Signature Hashing Algorithms: Three leaf certificates signed using either MD4, MD5, or SHA1.

Before using these faulty certificates, we assess them against the latest version of Firefox (v53.0, at the time of testing), and verify that Firefox terminates all TLS connections involving these certificates. As part of the analysis of the certificate validation mechanisms, we ensure that the TLS proxies do not cache TLS certificates, by checking the 'Organization Name' field of the subject parameter in the server certificates. Each leaf certificate of the crafted chains contains a unique 'Organization Name' value, allowing us to identify exactly which TLS certificate is being proxied.

4.8 Private Key Protection, Self-issued, and Pre-Generated Certificates

We attempt to locate a TLS proxy's private key (corresponding to its root certificate), and learn if it is protected adequately, e.g., inaccessible to non-root processes, encrypted under an admin password. Subsequently, we use the located private keys to sign leaf certificates, and check if the TLS proxy accepts its own certificates as the issuing authority for externally delivered content.

To locate the private keys on the non-Windows systems, access to the network appliances' disks content and their filesystems is required. If we get access to an appliance's filesystem (following Section 4.2), we search for files with the following known private key file extensions: ".pem", ".key", ".pfx", and ".p12", and then compare the modulus of located RSA private keys with the proxy's public key certificate to locate the correct corresponding key. If the filesystem is inaccessible, we parse the raw disks for keys, using the Linux command *strings* on the virtual hard disk file and search for private keys. We also use memory analysis tools, such as Volatility [23] and Heartleech [9], to extract the private keys in some cases; for more, see Appendix B. If we acquire the private key using this methodology, we still get no information on the key's location within the appliance's file system, storage method (e.g. encrypted, obfuscated), and privileges required to access the key. For Windows-based appliances, we utilize Mimikatz [15] to extract the private key (cf. [32]). Key storage is usually handled on Windows using two APIs: Cryptography API (CAPI), or Cryptography API: Next Generation (CNG [24]). When executed with Administrator privileges, Mimikatz exports private keys that are stored using CAPI and CNG. We check the location of the private keys, the privilege required to access them and if any encryption or obfuscation is applied.

We also check if appliance vendors rely on pre-generated certificates for their proxies, which could be very damaging. We install two instances of the same product, and compare the certificates

Table 2: Results for TLS parameter mapping/mirroring and vulnerabilities to known attacks. For TLS version mapping, we display the TLS versions seen by the client when the web server uses TLS 1.2, 1.1, 1.0 and SSL 3.0 ('–' means unsupported). Under "Problematic Ciphers": Weak means deprecated; Insecure means broken; blank means good ciphers. Under "Patched Against Attacks": ✗ means vulnerable; ✗* means potentially vulnerable (unknown if CBC is patched with $1/(n-1)$ split); blank means patched.

	TLS Version Mapping				Cipher Suites		Certificate Parameter Mapping			Patched Against Attacks				
	TLS 1.2	TLS 1.1	TLS 1.0	SSL 3.0	Cipher Suites Mirroring	Problematic Ciphers	Key Length Mapping	Signature Algorithm Mapping	EV Certificates	BEAST	FREAK	Logjam	CRIME	Insecure Renegotiation
Untangle	1.2	1.2	1.2	–	✗	Weak	2048	SHA256	DV	✗*				
pfSense	1.2	1.2	–	–	✗		4096	SHA256	DV					
WebTitan Gateway	1.2	1.2	1.2	–	✗	Weak + Insecure	1024	SHA256	DV	✗*			✗	
Microsoft TMG	–	–	1.0	3.0	✗	Weak + Insecure	2048	Mirrored	DV	✗				
UserGate Web Filter	1.2	1.2	1.2	–	✗	Weak + Insecure	1024	SHA256	DV	✗*				
Cisco Ironport WSA	1.2	1.2	1.2	–	✗		2048	SHA256	DV	✗*				

along with their correspondent private keys (if located). If we find the same key, we conclude that the appliance uses a pre-generated certificate, instead of per-installation keys/certificates.

5 RESULTS

In this section, we present the results of our analysis.

5.1 TLS Parameters

See Table 2 for an overview.

TLS versions and mapping. Four appliances support TLS versions 1.2, 1.1, and 1.0; pfSense does not support TLS 1.0. Microsoft Threat Management Gateway (TMG) supports only TLS 1.0 and (more worryingly) SSLv3; as many web servers nowadays do not support these versions (specifically SSLv3), we suspect that clients behind TMG will be unable to visit these websites. Except TMG, other appliances map all the proxy-to-server TLS versions to TLS 1.2 for the client-to-proxy connection (i.e., artificial version upgrade, misleading browsers/users).

Certificate parameters and mapping. No appliance mirrors the RSA key sizes; instead, they use hard-coded key lengths for all generated certificates. Two use non-recommended RSA-1024 certificates, while the remaining four artificially upgrade/downgrade the external key-length, and thus may mislead clients/users. Regarding hashing algorithms used for signing certificates, five appliances use SHA256 and thus will make external SHA1-based certificates (considered insecure) invisible to browsers; only Microsoft TMG mirrors the hash algorithm. All appliances intercept TLS connections with EV certificates, and thus, inevitably downgrade any EV certificate to DV (as the proxies cannot generate EV certificates).

Cipher suites. We use the Qualys Client Test [18] to determine the list of cipher suites used by the TLS proxies; no proxy mirrors the cipher suites and use a hard-coded list instead.

The Cisco Ironport WSA and pfSense exclude any weak or insecure ciphers from their cipher suites. Untangle, WebTitan Gateway, Microsoft TMG, and UserGate offer 3DES, which is now considered weak due its relatively small block size [28]. UserGate offers the insecure DES cipher [47]. Microsoft TMG and WebTitan include RC4, which has been shown to have biases [48], and is no longer supported by any modern browsers. Microsoft TMG additionally includes the deprecated MD5 hash algorithm [49], and insecure SSLv3 ciphers. WebTitan Gateway includes an IDEA cipher [29] with a 64-bit block length. When relying on the DHE ciphers, a reasonably secure modulus value should be used, e.g., 2048 or higher [26]. All of the tested appliances accepted a modulus size of 1024-bit; UserGate even accepted a 512-bit modulus.

Known TLS attacks. pfSense does not support TLS 1.0 by default, so it is considered as safe against BEAST. For a system to be vulnerable to BEAST, it has to support TLS 1.0, and use the CBC mode. However, after the BEAST attack was uncovered, a patch was released for CBC, but was identically named as CBC. Because of that, there is no easy way to distinguish between the unpatched CBC and patched CBC (implementing the $1/(n-1)$ split patch [1], initially pushed by Firefox). Untangle, WebTitan Gateway, UserGate, and Cisco Ironport WSA are possibly vulnerable to BEAST. However, Microsoft TMG is vulnerable to the BEAST attack, as its CBC cipher was recognized by howsmyssl.com. When combined with a Java applet to bypass the same origin policy, the BEAST attack could leak authentication cookies. Only WebTitan Gateway is vulnerable to the CRIME attack due to its support for TLS compression [34]. All appliances are patched against the FREAK and Logjam attacks, and use a secure renegotiation.

5.2 Certificate Validation Results

In this section, we discuss the vulnerabilities found in the certificate validation mechanism of the tested TLS proxies; for summary, see Tables 3, 4 and 5.

WebTitan and UserGate do not perform *any* certificate validation; their TLS proxies allowed *all* the faulty TLS certificates. This can cause a trivial MITM attack; attackers can simply use a self-signed certificate for any desired domain, fooling even the most secure and up-to-date browsers behind WebTitan and UserGate. In addition, UserGate caches certificates and does not reconsider changes in the server-side certificate. Both appliances however block access to the web pages that utilize RSA-512 server certificates, possibly triggered by the TLS libraries utilized by the TLS proxies, and not the TLS interception certificate validation code (as apparent from the error messages we observed). Therefore, we omit WebTitan Gateway and UserGate from the remaining discussion here.

All six appliances accept certificate chains with intermediates that have a bad name constraint. Untangled forwards the wrong CN

Table 3: Results for certificate validation, part I. ✓ means a faulty certificate is accepted and converted to a valid certificate by the TLS proxy; → means a faulty certificate is passed to the client as-is but not blocked; and blank means certificate blocked.

	Self Signed	Signature Mismatch	Fake Geo-Trust	Wrong CN	Unkn-own Issuer	Non-CA Interm-ediate	X509v1 Interm-ediate	Invalid pathLen-Constraint	Bad Name Constraint Intermediate	Unknown Critical X509v3 Extension	Malformed Extension Values
Untangle				→					✓		✓
pfSense									✓		→
WebTitan Gateway	✓	✓	✓	→	✓	✓	✓	✓	✓	✓	→
Microsoft TMG									✓		✓
UserGate Web Filter	✓	✓	✓	→	✓	✓	✓	✓	✓	✓	→
Cisco Ironport WSA									✓		→

Table 4: Results for certificate validation, part II. For legend, see Table 3; N/A means not tested as the appliance disallows adding the corresponding faulty CA certificate to its trusted store.

	Revoked	Expired Leaf	Expired Interm-ediate	Expired Root	Not Yet Valid Leaf	Not Yet Valid Interm-ediate	Not Yet Valid Root	Leaf keyUsage w/out Key Enciph-erment	Root keyUsage w/out KeyCert-Sign	Leaf extKey-Usage w/ clientAuth	Root extKey-Usage w/ Code Signing
Untangle	✓			✓			✓		✓		✓
pfSense	✓										✓
WebTitan Gateway	✓	→	✓	✓	→	✓	✓	→	✓	→	✓
Microsoft TMG								✓			✓
UserGate Web Filter	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Cisco Ironport WSA		→		N/A			N/A		N/A		N/A

certificates to the browser, while pfSense, Microsoft TMG and Cisco Ironport WSA terminate the connection. Regarding malformed extension values, none of the appliances blocked the anomalous certificate: Untangle and Microsoft TMG accepted it and displayed the web page, while pfSense and Cisco Ironport WSA passed it to the browser, which caught it and blocked access. Microsoft TMG and Cisco Ironport WSA check the revocation status of the offered certificates, while Untangle and pfSense do not. When exposed to expired leaf certificates, Untangle, pfSense and Microsoft TMG block access, while Cisco Ironport WSA forwards the certificates to the browser, as its default settings are configured to only monitor expired leaf, and not to drop the connections.

Untangle fails to detect expired or not yet valid root CA certificates; pfSense and Microsoft TMG block access, while Cisco Ironport WSA disallows adding them to its trusted store in the first place. When exposed to leaf certificates whose keyUsage do not include keyEnciphernment, Untangle, pfSense and Cisco Ironport WSA terminate the TLS connections, while Microsoft TMG allows them. Untangle fails to detect root CA certificates that do not have keyCertSign among the keyUsage values, while pfSense and Microsoft TMG block access, and Cisco Ironport WSA disallows adding such a root CA certificate to its trusted store. Similarly, Cisco Ironport WSA disallows adding root CA certificates whose extKeyUsage parameter is codeSigning, while Untangle, pfSense and Microsoft TMG accept connections with such certificates.

As for root CA RSA key sizes, Cisco Ironport WSA does not allow adding RSA-512 root certificates to its store, Untangle and pfSense permit TLS connections involving such certificates, while Microsoft TMG successfully terminates the connection. All of the tested appliances permit server certificates signed by RSA-1024 root CA certificates.

Regarding leaf certificate key sizes, all appliances terminate connections that utilize RSA-512. Untangle, pfSense and Cisco Ironport WSA accept RSA-768, RSA-1016, and RSA-1024 certificates, while Microsoft TMG blocks access to RSA-768 and RSA-1016 but allows RSA-1024 certificates. Microsoft TMG mirrors signature hashing algorithms, so weak and deprecated hash algorithms are passed to the client. Untangle and pfSense do not accept certificates signed using the MD4 algorithm, while Cisco Ironport WSA continues connections with such certificates. On the other hand, MD5 and SHA1 are accepted by pfSense and Cisco Ironport WSA, while Untangle accepts SHA1 and disallows MD5. When exposed to certificates signed by their own key pair, Untangle and Microsoft TMG fail to notice the anomaly, while pfSense and Cisco Ironport WSA terminate the connection. Note that, when a TLS connection is terminated, Untangle and Microsoft TMG use a TLS handshake failure; pfSense redirects the connection to an error page; and Cisco Ironport WSA uses an untrusted CA certificate, relying on the browser to block it.

For all appliances, we check if the TLS inspection feature is enabled by default after the activation of the appliances, or if it requires any manual step. Only UserGate Web Filter enables inspection by default. Thus, due to the lack of any certificate validation in UserGate, users located behind a freshly installed UserGateappliance are automatically vulnerable to trivial MITM attacks.

Table 5: Results for certificate validation, part III. For legend, see Table 3.

	Root Key Length (Good Leaf)		Leaf Key Length (Good Root)				Signature Hashing Algorithm			DHE Modulus Length		Own Root
	512	1024	512	768	1016	1024	MD4	MD5	SHA1	512	1024	
Untangle	✓	✓		✓	✓	✓			✓		✓	✓
pfSense	✓	✓		✓	✓	✓		✓	✓		✓	
WebTitan Gateway	✓	✓		✓	✓	✓	✓	✓	✓		✓	✓
Microsoft TMG		✓				✓	→	→	→		✓	✓
UserGate Web Filter	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
Cisco Ironport WSA	N/A	✓		✓	✓	✓	✓	✓	✓		✓	

Table 6: Results for CA trusted stores, private keys and initial setup. 'N/A': not available as we could not locate the private key on disk; 'None': the appliance does not rely on any store, as it performs no certificate validation; 'World': readable by any user account on the appliance.

	Trusted CA Store		Private Key			Initial Behavior	
	Location	Type	Location	State	Read Permission	Inspection By Default	Pre-Generated Key Pair
Untangle	/usr/share/untangle/lib/ ssl-inspector/trusted-ca-list.jks	Java Key Store	/usr/share/untangle /settings/untangle-certificates/untangle.key	Plaintext	Root	Off	No
pfSense	/usr/local/share /certs/ca-root-nss.crt	Mozilla NSS	/usr/local/etc /squid/serverkey.pem	Plaintext	World	Off	No
WebTitan Gateway	N/A	None	/usr/blocker/ssl /ssl_cert/webtitan.pem	Plaintext	World	Off	No
Microsoft TMG	mmc.exe Windows Trusted Store Local Computer	Microsoft Store	CERT_SYSTEM_STORE _LOCAL_MACHINE_MY	Exportable Key	Admin	Off	No
UserGate Web Filter	N/A	None	/opt/entensys/webfilter /etc/private.pem	Plaintext	World	On	No
Cisco Ironport WSA	Network ⟶ Certificate Management ⟶ Cisco Trusted Root Certificate List	Cisco GUI	N/A	N/A	N/A	Off	No

5.3 CA Trusted Stores

In this section, we analyze the results concerning CA trusted stores, their accessibility, source, and content; see Table 6. Note that, as WebTitan Gateway and UserGate Web Filter do not perform any certificate validation, their trusted stores are of no use.

Accessibility. Untangle's file system can be accessed through SSH. We found that Untangle relies on two CA trusted stores, saved in Java Keystore files on the filesystem. The first store, 'trusted-ca-list.jks', holds the CA authorities trusted by default by Untangle, while the second store, 'trustStore.jks', holds the custom CA certificates, added by the machine administrator through Untangle's UI. pfSense also allows SSH, and we found that its CA trusted store on the FreeBSD filesystem under 'ca-root-nss.crt'. pfSense does not offer any UI to add custom CA certificates. We append our crafted certificates to the original store, in a format that includes the public key, in addition to the text meta-data (OpenSSL's '-text' option). Microsoft TMG relies on the Windows Server's standard trusted store. To view the content of the trusted store and to inject our crafted CA certificates, we rely on the conventional tool, the Microsoft Management Console, in the Trusted Root Certification Authorities section of the Local Computer. Cisco Ironport WSA's trusted CA store can be accessed through the appliance's web interface, under the Cisco Trusted Root Certificate List. The appliance also includes another interface, the Cisco Blocked Certificates, for untrusted issuer certificates. To add custom CA certificates, the appliance includes a third interface, the Custom Trusted Root Certificates. However, Cisco Ironport WSA does not allow the injection of most of the invalid Root certificates, and responds with an error in these cases.

Source and content. As documented on Untangle's SSL Insepctor wiki page [21], the list of trusted certificates is generated from the Linux ca-certificates package, in addition to Mozilla's list of certificates. However, the list includes Microsoft's own Root Agency certificate, which indicates the additional inclusion of the Windows trusted certificates. The Root Agency certificate is an RSA-512 certificate, valid until 2039, which can be readily compromised.[1] It has been used since the 1990s as the default test certificate for code signing and development; Windows systems still include this certificate, but mark it as untrusted. The private key corresponding to this certificate could also be extracted from the MakeCert tool, as it is hard-coded. As a result, Untangle is instantly vulnerable to a trivial MITM attack, using the Root Agency's certificate key pair.

[1]In 2016, it took about four hours and $75 to factor RSA-512 using Amazon EC2 [46].

Untangle's trusted store also includes 21 RSA-1024 root certificates, and 30 expired certificates. It also includes certificates from issuers that are no longer trusted by major browser/OS vendors: three CNNIC ROOT CA certificates, six DigiNotar root CA certificates, three TÜRKTRUST CA certificates, and four WoSign CA certificates.

pfSense's trusted CA store relies on the Mozilla's NSS certificates bundle, extracted from the nss-3.30.2 version, with 20 untrusted certificates omitted from the bundle, as specified in the header of the trusted store. It does not include any RSA-512 or RSA-1024 certificates, and no expired certificates. However, the store includes two CNNIC ROOT CA certificates, and four WoSign CA certificates. Similar to all Windows operating systems, the Windows Server 2008 R2 which hosts Microsoft TMG, does not display the full list of trusted root certificates in its management console, and instead, only displays the root certificates of web servers already visited. We thus rely on the Microsoft Trusted Root Certificate Program [14] to browse the list of certificates trusted to the date of the testing. We found that the list includes two CNNIC ROOT CA certificates, two TÜRKTRUST CA certificates, two ANSSI CA certificates, and four WoSign CA certificates. Nonetheless, the acquired list does not include the RSA key sizes of the certificates, their expiry dates, or their revocation states. As for Cisco Ironport WSA, we found four potentially problematic root CA certificates included into the trusted store, related to TÜRKTRUST. All certificates included in Cisco's trusted store are trusted by current browser/OS vendors. However, the RSA key sizes are not displayed within the UI, so we could not check for RSA-512 and RSA-1024 CA certificates.

5.4 Private Key Protection

In this section, we discuss the results regarding the TLS proxies' private keys, in terms of storage location, state, and the privilege required to access them; see Table 6.

We could not access the filesystem of Cisco Ironport WSA's AsyncOS to locate the private key on disk. Instead, we extract its private key from memory using Heartleech [9] (see Appendix B for details). As there is no information on the private key on disk, and the located key was used only for testing external content signed by own key, we do not discuss this appliance in the rest of the section.

We rely on the methodologies stated in Section 4.2 to access the filesystem for non-Windows network appliances. pfSense and Untangle's access is acquired through SSH, while WebTitan Gateway's access is acquired through the mounting of its virtual disk drive on a separate machine. UserGate provides access to its OS terminal by default. Untangle, pfSense, WebTitan Gateway and UserGate store their plaintext private keys in directories on their OS filesystems (as 'untangle.key', 'serverkey.pem', 'webtitan.pem', and 'private.pem' files, respectively). pfSense, WebTitan Gateway and UserGate allow the read file permission to all users accounts (write is restricted to root), while Untangle allows read/write only to the root account.

Microsoft TMG's private key is stored using Microsoft's Software Key Storage Provider, utilizing CNG. The key is an exportable key, meaning that it can be exported through the Microsoft Management Console, if opened with the right privileges. The Microsoft Management Console must be executed with SYSTEM privileges to export the private key. We rely on the Mimikatz tool to export the key, which requires a less privileged Administrator account.

None of the tested appliances use pre-generated keys/certificates. We installed multiple instances of each appliance, and each one has a distinct certificate.

5.5 Practical Attacks

In this section, we summarize how the vulnerabilities reported could be exploited by an attacker.

MITM attacks can be trivially launched to impersonate any web server against clients behind UserGate (by default) and WebTitan Gateway (after TLS interception is enabled), due to their lack of certificate validation. Clients behind Untangle are also similarly vulnerable, due to the RSA-512 'Root Agency' certificate in its trusted store.

UserGate, WebTitan Gateway, Microsoft TMG, and Untangle accept external ceritificates signed their own root key. If an attacker can gain access to the signing keys of these appliances, she can launch MITM attacks to impersonate any web server. UserGate, and WebTitan Gateway provide 'read' privileges to non-root users for the private signing key, while Untangle and Microsoft TMG make it harder for attackers that have compromised the appliances, requiring root/admin privileges to access the keys.

An attacker can recover authentication cookies from the clients behind WebTitan Gateway by exploiting the CRIME [34] vulnerability, and the clients behind Microsoft TMG by exploiting the BEAST [33] vulnerability. Attackers could also recover cookies from clients behind WebTitan Gateway and Microsoft TMG due to their use of RC4 [48]

We contacted the 6 affected companies and received replies from three companies; Untangle replied with just an automatic reply, Entensys confirmed that they have passed the matter to its research team. Netgate (pfSense) stated that they philosophically oppose TLS interception, but include it as it is a commonly requested feature. Netgate also states that the TLS interception is done using the external package 'Squid', which it does not control completely. .

6 CONCLUSION

We present a framework for analyzing TLS interception behaviors of network appliances to uncover any potential vulnerabilities introduced by them. We tested six network appliances, and found that all their TLS proxies are vulnerable against the tests under our framework—at varying levels. Each proxy lacks at least one of the best practices in terms of protocol and parameters mapping, patching against known attacks, certificate validation, CA trusted store maintenance, and private key protection.

We found that the clients behind four appliances are vulnerable to full server impersonation under an active man-in-the-middle attack, of which one enables TLS interception by default; and that attackers can recover authentication cookies with the current mechanism of two security appliances. Furthermore, client browsers are often being misled, as the TLS versions and certificate parameters displayed have a higher security level than the actual proxy-to-server TLS connection, similar to client-end TLS proxies [32].

While TLS proxies are mainly deployed in enterprise environments to decrypt the traffic in order to scan it for malware and network attacks, the network appliances introduce new intrusion opportunities and vulnerabilities for attackers. As TLS proxies act as the client for the proxy-to-web server connections, they should maintain (at least) the same level of security as modern browsers; similarly, as they act as a TLS server for the client-to-proxy connections, they should be securely configured like any up-to-date HTTPS server, by default. Before enabling TLS interception, concerned administrators may use our framework to evaluate their network appliances, and consider the potential vulnerabilities that may be introduced by a TLS proxy against its perceived benefits.

ACKNOWLEDGEMENTS

We thank the ASIA CCS 2018 anonymous reviewers for their helpful comments and suggestions to improve the paper's presentation, and Xavier de Carné de Carnavalet for his help during the development of our framework. This research is supported by NSERC.

REFERENCES

[1] BEAST attack 1/n-1 split patch. Available at https://goo.gl/8MYeqz.
[2] Cisco WSA AsyncOS documentation. Available at https://goo.gl/hkHRbK.
[3] Diginotar ca breach. Available at https://goo.gl/p9ainQ, Sep 2011.
[4] Distrusting new CNNIC certificates. Available at https://goo.gl/yPidqC, Apr 2015.
[5] Distrusting new WoSign and StartCom certificates. Available at https://goo.gl/zGmf5b, Oct 2016.
[6] Effects of HTTPS and SSL inspection on the client. Available athttps://goo.gl/q1MVw4, Aug 2017.
[7] Extended validation OID. Available at https://goo.gl/AmmnXE, Oct 2013.
[8] GRC certificate validation revoked test, note=Available at https://goo.gl/A83vCC.
[9] Heartleech - GitHub. Available at https://goo.gl/JeKcpt.
[10] Howsmyssl - GitHub. Available at https://goo.gl/48gyGd.
[11] Lenovo's superfish security. Available at https://goo.gl/w2R2y5, Feb 2015.
[12] Microsoft TMG 2010 updates. Available at https://goo.gl/WcykM6.
[13] Microsoft TMG supported OS version. Available at https://goo.gl/SU9LQ8.
[14] Microsoft trusted root certificate program. Available at https://goo.gl/5BT7d8.
[15] Mimikatz - GitHub. Available at https://goo.gl/dUWCmH.
[16] Revoking ANSSI CA. Available at https://goo.gl/rCjwtY, Dec 2013.
[17] The risks of SSL inspection. Available athttps://goo.gl/S3mL5v, Mar 2015.
[18] SSL client test. Available at https://goo.gl/3RdQ1J.
[19] The TÜRKTRUST SSL certificate fiasco. Available at https://goo.gl/8gxCdc, Jan 2013.
[20] UFS - Linux Kernel archives. Available at https://goo.gl/yZ3Fty.
[21] Untangle SSL inspector documentation. Available at https://goo.gl/NZghGy.
[22] US-CERT alert on HTTPS interception. Available at https://goo.gl/9oqZ4w.
[23] Volatility. Available at https://goo.gl/LSnbwF.
[24] Windows cryptography API (CNG). Available at https://goo.gl/UrARyq.
[25] ZMap - GitHub. Available at https://goo.gl/1g2UtU.
[26] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, Denver, CO, USA, 2015.
[27] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue. A messy state of the union: Taming the composite state machines of tls. In *2015 IEEE Symposium on Security and Privacy*, pages 535–552, Fairmont, CA, USA, 2015.
[28] K. Bhargavan and G. Leurent. On the practical (in-) security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 456–467, Vienna, Austria, 2016.
[29] E. Biham, O. Dunkelman, N. Keller, and A. Shamir. New attacks on IDEA with at least 6 rounds. *Journal of Cryptology*, 28(2):209–239, 2015.
[30] C. Brubaker, S. Jana, B. Ray, S. Khurshid, and V. Shmatikov. Using Frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *2014 IEEE Symposium on Security and Privacy*, pages 114–129, Fairmont, CA, USA, 2014.
[31] S. Y. Chau, O. Chowdhury, E. Hoque, H. Ge, A. Kate, C. Nita-Rotaru, and N. Li. SymCerts: Practical symbolic execution for exposing noncompliance in x.509 certificate validation implementations. In *2017 IEEE Symposium on Security and Privacy*, pages 61–68, Fairmont, CA, USA, 2017.
[32] X. de Carné de Carnavalet and M. Mannan. Killed by proxy: Analyzing client-end tls interception software. In *Network and Distributed System Security Symposium*, San Diego, CA, USA, 2016.
[33] T. Duong and J. Rizzo. Here come the ⊕ ninjas. *Technical Report*. Available at https://goo.gl/DujxQg, May 2011.
[34] T. Duong and J. Rizzo. The CRIME attack. *Presentation at Ekoparty Security Conference*, 2012.
[35] Z. Durumeric, Z. Ma, D. Springall, R. Barnes, N. Sullivan, E. Bursztein, M. Bailey, J. A. Halderman, and V. Paxson. The security impact of HTTPS interception. In *Network and Distributed Systems Symposium*, San Diego, CA, USA, 2017.
[36] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. Why eve and mallory love android: An analysis of Android SSL (in) security. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 50–61, Raleigh, NC, USA, 2012.
[37] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49, Raleigh, NC, USA, 2012.
[38] B. He, V. Rastogi, Y. Cao, Y. Chen, V. Venkatakrishnan, R. Yang, and Z. Zhang. Vetting SSL usage in applications with SSLint. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534, Fairmont, CA, USA, 2015.
[39] R. Housley, W. Ford, W. Polk, and D. Solo. RFC 5280: Internet x.509 public key infrastructure certificate and crl profile, May 2008.
[40] L. S. Huang, A. Rice, E. Ellingsen, and C. Jackson. Analyzing forged SSL certificates in the wild. In *2014 IEEE Symposium on Security and Privacy*, pages 83–97, Fairmont, CA, USA, 2014.
[41] J. Jarmoc. SSL/TLS interception proxies and transitive trust. *Black Hat Europe*, Mar 2012.
[42] M. O'Neill, S. Ruoti, K. Seamons, and D. Zappala. TLS proxies: Friend or foe? In *Proceedings of the 2016 ACM on Internet Measurement Conference*, pages 551–557, Santa Monica, CA, USA, 2016.
[43] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov. RFC 5746: Transport layer security (tls) renegotiation indication extension, Feb 2010.
[44] S. Ruoti, M. O'Neill, D. Zappala, and K. E. Seamons. User attitudes toward the inspection of encrypted traffic. In *Proceedings of the Eleventh Symposium On Usable Privacy and Security*, pages 131–146, Denver, CO, USA, 2016.
[45] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana. HVLearn: Automated black-box analysis of hostname verification in SSL/TLS implementations. In *2017 IEEE Symposium on Security and Privacy*, pages 521–538, Fairmont, CA, USA, 2017.
[46] L. Valenta, S. Cohney, A. Liao, J. Fried, S. Bodduluri, and N. Heninger. Factoring as a service. In *International Conference on Financial Cryptography and Data Security*, pages 321–338, Christ Church, Barbados, 2016.
[47] P. Van De Zande. The day DES died. *SANS Institute*, Jul 2001.
[48] M. Vanhoef and F. Piessens. All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS. In *USENIX Security Symposium*, pages 97–112, Washington D.C., USA, 2015.
[49] X. Wang and H. Yu. How to break MD5 and other hash functions. In *37th Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 19–35, Sofia, Bulgaria, 2005.

A CRAFTING INVALID CERTIFICATE CHAINS

In this section, we detail the methodology used to create each certificate validation test.

Self-Signed. We generate a standalone certificate using OpenSSL with regular parameters.

Signature Mismatch. We first generate a regular CA certificate, and use it to sign a regular leaf certificate. We then modify the signature of the leaf public key certificate by flipping one of the last bits in the certificate. The certificate signature is positioned as the last item inside the certificate. We thus create a certificate with a mismatching signature, and test if the proxy validates the signature on the presented certificate.

Fake GeoTrust Global CA. We craft an issuing root certificate with the same certificate parameters as the GeoTrust Global CA authority. We mimic the Common Name field (CN = GeoTrust Global CA), the Organization field (O = GeoTrust Inc.), and the Country field (C = US). Before signing the leaf certificate, we remove the authority key identifier parameter from it. Without the authority key identifier, the certificate cannot be linked to its issuing certificate.

By doing so, we test if the TLS proxy validates the chain of trust properly, or relies only on the certificate parameters such as the subject name alone.

Wrong Common Name (CN). We generate a regular root CA certificate, and use it to sign a regular leaf certificate that does not have *apache.host* as the value for Common Name field. If the TLS proxy accepts such a leaf certificate for the *apache.host* domain, then the proxy does not validate that the delivered certificate is for the exact domain requested, and thus, allows websites to impersonate other servers by using their valid certificate.

Unknown Issuer. The test relies on a normal issuing certificate and its normal leaf certificate. However, we do not import the issuing certificate to the trust store of the network appliance, and consequently, check if the TLS proxy is vulnerable to MITM attacks, when an attacker uses untrusted CA certificates as issuers for their certificate.

Non-CA Intermediate. We generate three certificates that serve respectively as the root CA, the intermediate certificate and the leaf certificate. However, we intentionally craft the intermediate certificate to have the basic constraint extension that identifies CA certificate as false. Using this methodology, we test if the TLS proxy ensures that the CA certificates have the ability to issue other certificates, using the CA flag. If the proxy does not detect such vulnerabilities, attackers could use any valid leaf certificate to sign other leaf certificates, and host them on their servers.

X509v1 Intermediate. The first version of x509 does not have the basic constraint extension, and thus, CA certificates cannot be differentiated from leaf certificates. As a result, x509v1 certificates should not be used for issuing certificates. We generate three certificates that serve respectively as the root CA, the intermediate certificate and the leaf certificate, while only having the intermediate certificate of type x509v1. If accepted, the proxy risks potential consequences that are similar to the Non-CA Intermediate test.

Revoked. We test if the TLS proxy accepts revoked certificates using Gibson Research Corporation's special site that hosts a website using a revoked certificate [8]. Digicert provided them with an intentionally revoked certificate using both a Certificate Revocation List (CRL) and the Online Certificate Status Protocol (OCSP). If the revoked certificate is allowed, this implies that the TLS proxy does not validate the revocation status of the delivered certificates and their appropriate issuers.

Expired and Not Yet Valid Certificates. We generate three distinct tests to check the behavior of network appliances when exposed to expired certificate. For the first test, we craft a root CA certificate with an expired validity date, and use it to sign a regular leaf certificate. For the second test, we craft a regular root CA certificate, use it to sign an expired intermediate certificate, which in turn, signs a regular leaf certificate. For the third test, we craft a regular root CA certificate, and use it to sign an expired leaf certificate. Similarly, we generate three similar tests for not yet valid certificates. The main difference between the two set of certificates is that, for expired certificates, the 'valid to' date is prior to the current date of testing, while the not yet valid certificates have the 'valid from' date exceeding the date of testing.

Invalid pathLenConstraint. A pathLenConstraint of 1 in a root CA certificate implies that the issuer can issue one layer of intermediate certificates, which in turn will have a pathLenConstraint of 0. A pathLenConstraint of 0 implies that this certificate can only issue leaf certificates. We generate a root CA certificate with a pathLenConstraint of 1, and issue an intermediate certificate with a pathLenConstraint of 0 using it, and subsequently issue another intermediate certificate with a pathLenConstraint of 0 using the first intermediate certificate. We then use the section intermediate certificate to issue a leaf certificate. Using this methodology, we test if the TLS proxies check the pathLenConstraint parameter, as the first intermediate should issue only leaf certificates, and not another intermediate certificate.

Bad Name Constraint Intermediate. We test if the TLS proxies validate the Name Constraint x509v3 certificate extension. We craft a regular CA certificate, and use it to sign an intermediate certificate that has a different domain than 'apache.host' solely permitted as a DNS name. We then issue a leaf certificate for the domain *apache.host* using that intermediate certificate. The validating proxy should typically terminate the TLS connection when exposed to such a case, as the intermediate certificate has an issuing permit constraint for a domain different than 'apache.host'.

Malformed X509v3 Extension Value. We generate a regular root CA certificate and use it to issue a leaf certificate that holds a dummy random string as a value for its keyUsage parameter (i.e., will not match any of the names in the list of the permitted key usages).

Unknown Critical X509v3 Extension. We generate a root CA certificate, and use it to issue a leaf certificate that holds a non-typical x509v3 extension (unusual OID value), set to critical. We thus analyze the TLS proxies' behavior when exposed to unknown extensions marked as critical.

Wrong keyUsage and extKeyUsage. We rely on two tests for the keyUsage and extKeyUsage x509v3 extensions, one for leaf certificates, and the other for root certificates. Regarding the keyUsage, we craft a regular root certificate, and sign a leaf certificate that holds a keyUsage value of keyCertSign, omitting the required keyEncipherment value for all leaf certificates. TLS proxies should drop TLS connection to servers that hold no keyEncipherment keyUsages. Moreover, we craft a root CA certificate with a keyUsage of nonRepudiation, omitting the required keyCertSign value for all issuing certificates. TLS proxies should not accept issuing certificates with a keyUsage value that excludes keyCertSign.

Regarding the extKeyUsage extension, we craft a regular root certificate and use it to issue a leaf certificate that holds clientAuth as the extKeyUsage value, implying that this certificate is meant to be used by TLS clients, and not by TLS servers. TLS proxies should then drop the TLS connection. We also craft a root certificate whose extKeyUsage value consists of codeSigning, implying that this issuing certificate is not meant to be used for TLS connections. TLS proxies should similarly drop such a connection. Failure of proper validation of certificate usages allows attackers to abuse TLS connections by using non-compliant certificates.

Short Key Length Root and Leaf Certificates. We generate RSA-512 and RSA-1024 root CA certificates, and import them to the trusted stores (when possible). We host their respective leaf certificates, and test if the TLS proxy accepts insecure key sizes for root certificates. On the other hand, we generate regular root certificates with proper key sizes (e.g., RSA-2048), and craft their leaf certificates to have short keys (512, 768, 1016 and 1024 as key sizes), and test if the TLS proxies accept such insecure key sizes for leaf certificates.

Bad Signature Hashing Algorithms. To check the proxies' behavior when exposed to weak and deprecated signature algorithms, we modify the signature algorithms in the OpenSSL configuration file when signing three distinct certificates to use respectively MD4, MD5 and SHA1.

B PRIVATE KEY EXTRACTION FOR CISCO IRONPORT WSA

In this section, we discuss the challenges faced while attempting to extract the private key from the Cisco Ironport Web Security Appliance.

We performed several attempts to bypass the limited custom command line interface, and access the filesystem content itself. We first tried to skip the proprietary command-line interface and reach operating system's native command-line. However, Cisco's interface is designed to have no escape point out of its custom command-line interface [2]. We then attempted to mount the network appliance's drive to our Ubuntu machine. We discovered that the virtual disk drive is divided into 9 different partitions, with FreeBSD as the main OS. We subsequently attempted to mount all partitions, with FreeBSD's filesystem type UFS, and all UFS type options, which include: old, default, 44bsd, ufs2, 5xbsd, sun, sunx86, hp, nextstep, nextstep-cd, and openstep [20].

With all the mentioned UFS types failing and no mounted drive, we then attempted to explore the content of the disk drive without mounting it. We relied on the Linux *strings* command, which extracts printable characters from binary files. We then parsed the output and saved all private keys found, by searching for the private key delimiters '`-----BEGIN PRIVATE KEY-----`' and '`-----BEGIN RSA PRIVATE KEY-----`'. We then compared the modulus of each key to the appliance's public key certificate, in order to attempt to locate the corresponding private key.

With no positive matching, we proceeded to memory analysis. We dumped the volatile memory of a VM by saving a snapshot of the running machine, after intercepting the traffic for a not previously visited website. We ensured that the website visited has not been visited and proxied earlier, to guarantee that the private key will be used to sign the intercepted page, and thus, be located in the appliance's RAM. Subsequently, we passed the memory dump to Volatility, a memory forensics tool [23]. Volatility requires as an input the exact profile of the OS corresponding to the memory dump. Consequently, we attempted to determine the profile using Volatility's 'imageinfo' command, which fails to determine the profile. Without the specific profile, Volatility fails to execute.

As a result, we attempted to use the collected memory dump with Heartleech [9]. We fed the tool with Cisco Ironport Web Security Appliance's memory dump, along with the TLS proxy's public key certificate to Heartleech, which successfully outputs the corresponding private key.