2014 IEEE Symposium on Security and Privacy

Analyzing Forged SSL Certificates in the Wild

Lin-Shung Huang*, Alex Rice†, Erling Ellingsen†, Collin Jackson*
*Carnegie Mellon University, {linshung.huang, collin.jackson}@sv.cmu.edu
†Facebook, {arice, erling}@fb.com

***Abstract*—The SSL man-in-the-middle attack uses forged SSL certificates to intercept encrypted connections between clients and servers. However, due to a lack of reliable indicators, it is still unclear how commonplace these attacks occur in the wild. In this work, we have designed and implemented a method to detect the occurrence of SSL man-in-the-middle attack on a top global website, Facebook. Over 3 million real-world SSL connections to this website were analyzed. Our results indicate that 0.2% of the SSL connections analyzed were tampered with forged SSL certificates, most of them related to antivirus software and corporate-scale content filters. We have also identified some SSL connections intercepted by malware. Limitations of the method and possible defenses to such attacks are also discussed.**

***Keywords*-SSL; certificates; man-in-the-middle attack;**

I. Introduction

Secure Socket Layer (SSL) [1], or its successor, Transport Layer Security (TLS) [2], is an encryption protocol designed to provide secure communication and data transfers over the Internet.[1] SSL allows clients to authenticate the identity of servers by verifying their X.509 [3] digital certificates, and reject connections if the server's certificate is not issued by a trusted certificate authority (CA). SSL is most popular for enabling the encryption of HTTP traffic between websites and browsers, but also widely used for other applications such as instant messaging and email transfers. An SSL man-in-the-middle attack is an interception of such an encrypted connection between a client and a server where the attacker impersonates the server through a *forged* SSL certificate — that is, an SSL certificate not provided or authorized by the legitimate owner. We explain how this is possible below.

In practice, certificates issued through hundreds [4] of CAs are automatically trusted by modern browsers and client operating systems. Under the current X.509 public key infrastructure, every single CA has the ability to issue trusted certificates to *any* website on the Internet. Therefore, CAs must ensure that trusted certificates are only issued to the legitimate owners of each website (by certifying the real identities of their customers). However, if any of the trusted CAs suffers a security breach, then it is possible for attackers to obtain forged CA certificates for any desired website. In other words, a single CA failure would allow the attacker to intercept all SSL connections on the Internet. In fact, multiple commercial CAs (DigiNotar [5], Comodo [6], and TURKTRUST [7]) have been found to mis-issue fraudulent certificates in the past. Some of these CA incidents actually resulted in real man-in-the-middle attacks against high-profile websites such as Google [8]. Since the attacker's certificates were signed by trusted CAs, standard browsers cannot simply distinguish the attacker's intercepting server from the legitimate server (unless the forged certificate is later revoked). Hypothetically [9], some governments may also compel CAs to issue trusted SSL certificates for spying purposes without the website's consent.

Furthermore, even if the attacker cannot obtain a trusted certificate of legitimate websites, it is still possible to intercept SSL connections against some users (that ignore browser security warnings). Historically, browsers tend to behave leniently when encountering errors during SSL certificate validation, and still allow users to proceed over a potentially insecure connection. One could argue that certificate warnings are mostly caused by server mis-configurations (e.g. certificate expirations) rather than real attacks, therefore browsers should let users determine whether they should dismiss the errors. However, designing an effective security warning dialog has been a challenging task for browser vendors. A number of usability studies [10], [11], [12], [13] have shown that many users actually ignore SSL certificate warnings. Note that users who incautiously ignore certificate warnings would be vulnerable to the simplest SSL interception attacks (using self-signed certificates).

Despite that SSL man-in-the-middle attack attempts have previously been observed in the wild (e.g. in Iran [8] and Syria [14]), it is unclear how prevalent these attacks actually are. Several existing SSL surveys [4], [15], [16], [17] have collected large amounts of SSL certificates via scanning public websites or monitoring SSL traffic on institutional networks, yet no significant data on forged SSL certificates have been publicly available. We hypothesize that real attackers are more likely to perform only highly targeted attacks at certain geographical locations, or on a small number of high-value sessions, therefore, previous methodologies would not be able to detect these attacks effectively.

Unfortunately, detecting SSL man-in-the-middle attacks from the website's perspective, on a large and diverse set of clients, is not a trivial task. Since most users do not use client certificates, servers cannot simply rely on SSL client authentication to distinguish legitimate clients from attackers. Furthermore, there is currently no way for a web application to check the certificate validation status of the underlying SSL connection, not even when an SSL error has occurred on the client. Also, it is currently not possible for web applications to directly access the SSL handshake with native browser networking APIs, like XMLHttpRequest and WebSockets, to validate SSL certificates on their own.

[1]For brevity, we refer to SSL/TLS as SSL in this paper.

DOI 10.1109/SP.2014.13

In this paper, we first introduce a practical method for websites to detect SSL man-in-the-middle attacks in a large scale, without alterations on the client's end (e.g. custom browsers). We utilized the widely-supported Flash Player plugin to enable socket functionalities not natively present in current browsers, and implemented a partial SSL handshake on our own to capture forged certificates. We deployed this detection mechanism on an Alexa top 10 website, Facebook, which terminates connections through a diverse set of network operators across the world. We analyzed $3,447,719$ real-world SSL connections and successfully discovered at least $6,845$ (0.2%) of them were forged SSL certificates.

Our contributions can be summarized as follows:

- We designed a novel method for websites to collect direct evidence of man-in-the-middle attacks against their SSL connections. We further implemented this detection method on Facebook's website.
- We conducted the first analysis on forged SSL certificates by measuring over 3 million SSL connections. Our results show that 0.2% SSL connections are in fact tampered with forged certificates.
- Based real-world data, we categorized the root causes of forged SSL certificates. We showed that most of the SSL interceptions are due to antivirus software and organization-scale content filters.
- We provided evidence of SSL interceptions by malware, which have infected users across at least 45 countries.

The rest of this paper is organized as follows. Section II provides background information and surveys related work. Section III details the design, implementation, and experimentation of our plugin-based detection method. Section IV gives an analysis of the forged SSL certificates that were observed. Section V surveys possible mitigations. Section VI concludes.

II. Background

In this section, we provide an overview of the SSL protocol, and how the SSL man-in-the-middle attack works. We then survey related work, and discuss existing tamper detection techniques that may be used by websites to detect network interceptions.

A. The SSL Protocol

The Secure Socket Layer (SSL) protocol was designed to ensure secure communications between two entities over untrusted networks. The SSL protocol provides authentication based on the X.509 public key infrastructure, protects data confidentiality using symmetric encryption, and ensures data integrity with cryptographic message digests. SSL is commonly used for securing websites and mail servers, preventing passive network attackers from eavesdropping or replaying the client's messages, and is generally considered security best practice for websites. By enabling encryption, websites can easily prevent the eavesdropping of unencrypted confidential data (e.g. Firesheep [18]).

To establish an SSL connection, the client and the server performs a handshake to authenticate each other, and negotiate

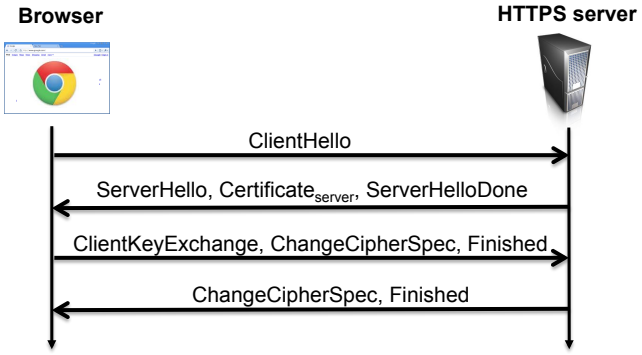

Fig. 1. A basic SSL handshake with no client certificates

the cipher algorithms and parameters to be used. Figure 1 depicts a basic SSL handshake using the RSA key exchange with no client certificates. First, the client sends a ClientHello message to the server, which specifies a list of supported cipher suites and a client-generated random number. Second, the server responds with the ServerHello message which contains the server-chosen cipher suite and a server-generated random number. In addition, the Certificate message contains the server's public key and hostname, digitally signed by a certificate authority, in which the client is responsible of verifying. The client then encrypts the pre-master secret using the server's public key and sends the pre-master secret to the server over a ClientKeyExchange message. Both the client and server can hence derive the same session key from the pre-master secret and random numbers. Finally, the client and server exchanges ChangeCipherSpec messages to notify each other that subsequent application data within the current session will be encrypted using the derived session key.

As mentioned in Section I, the SSL protocol allows clients to authenticate the identity of servers by verifying their SSL certificates. In practice, commercial SSL certificates are often signed by intermediate CAs (a delegated certificate signer), instead of directly signed by a trusted root CA (which are kept offline to reduce the risk of being compromised). Therefore, the server's Certificate message normally includes a chain of certificates, consisting of one leaf certificate (to identify the server itself), and one or more intermediate certificates (to identify the intermediate CAs). Each certificate is cryptographically signed by the entity of the next certificate in the chain, and so on. A valid certificate chain must chain up to a root CA that is trusted by the client. Note that SSL certificates are by design transferred in plaintext since the integrity can be verified by signatures. It is critical that clients must validate every certificate in the chain. In the following section, we will explain why validating SSL server certificates is necessary.

B. The SSL Man-in-the-Middle Attack

The SSL man-in-the-middle (MITM) attack is a form of active network interception where the attacker inserts itself into the communication channel between the victim client and

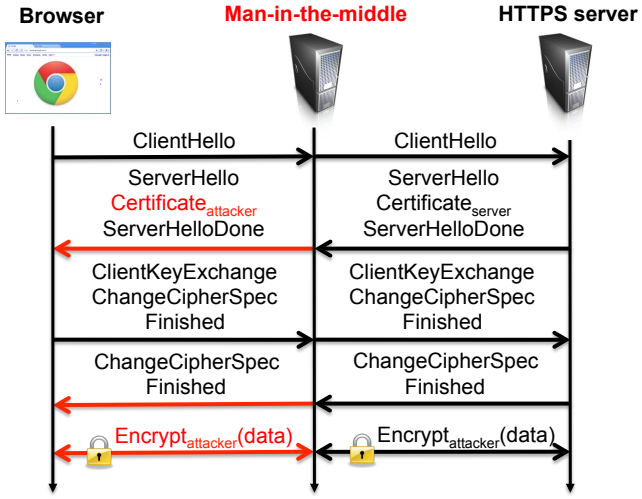

Fig. 2. An SSL man-in-the-middle attack between the browser and the server, using a forged SSL certificate to impersonate as the server to the client.

the server (typically for the purpose of eavesdropping or manipulating private communications). The attacker establishes two separate SSL connections with the client and the server, and relays messages between them, in a way such that both the client and the server are unaware of the middleman. This setup enables the attacker to record all messages on the wire, and even selectively modify the transmitted data. Figure 2 depicts an SSL man-in-the-middle attack with a forged certificate mounted between a browser and a HTTPS server. We describe the basic steps of a generic SSL man-in-the-middle attack as follows:

1) The attacker first inserts itself into the transport path between the client and the server, for example, by setting up a malicious WiFi hotspot. Even on otherwise trusted networks, a local network attacker may often successfully re-route all of the client's traffic to itself using exploits like ARP poisoning, DNS spoofing, BGP hijacking, etc. The attacker could also possibly configure itself as the client's proxy server by exploiting auto-configuration protocols (PAC/WPAD) [19]. At this point, the attacker has gained control over the client's traffic, and acts as a relay server between the client and the server.
2) When the attacker detects an SSL ClientHello message being sent from the client, the attacker accurately determines that the client is initiating an SSL connection. The attacker begins the impersonation of the victim server and establishes an SSL connection with the client. Note that the attacker uses a forged SSL certificate during its SSL handshake with the client.
3) In parallel to the previous step, the attacker creates a separate SSL connection to the legitimate server, impersonating the client. Once both SSL connections are established, the attacker relays all encrypted messages between them (decrypting messages from the client, and then re-encrypting them before sending to the server). Now, the attacker can read and even modify the encrypted messages between the client and the server.

As soon as the client accepts the forged SSL certificate, the client's secrets will be encrypted with the attacker's public key, which can be decrypted by the attacker. Note that regardless of whether the attacker's forged certificate is issued by a trusted CA, the attack steps are the same. If one of the client's trusted CAs went rogue or was otherwise coerced [9] into issuing a certificate for the attacker, the browser will automatically accept the forged certificate. In fact, professional attackers have proven capable of compromising CAs themselves in order to obtain valid certificates, as has occurred during the security breaches of DigiNotar [5] and Comodo [6]. Moreover, even if the attacker does not have a trusted certificate of the victim server and uses a self-signed certificate, researchers have shown that many users ignore SSL certificate warnings presented by the browser [11]. Even worse, studies have discovered that some non-browser software and native mobile applications actually contain faulty SSL certificate validation code, which silently accepts invalid certificates [20], [21], [22].

Lastly, numerous automated tools that can mount SSL man-in-the-middle attacks are publicly available on the Internet (e.g. sslsniff [23]), which greatly reduce the level of technical sophistication necessary to mount such attacks.

C. Certificate Observatories

A number of SSL server surveys [4], [15], [16], [17] have analyzed SSL certificates and certificate authorities on the Internet. The EFF SSL Observatory [4] analyzed over 1.3 million unique SSL certificates by scanning the entire IPv4 space, and indicated that 1,482 trusted certificate signers are being used. Similarly, Durumeric et al. [17] collected over 42 million unique certificates by scanning 109 million hosts, and identified 1,832 trusted certificate signers. Holz et al. [15] analyzed SSL certificates by passively monitoring live SSL traffic on a research network in addition to actively scanning popular websites, and found that over 40% certificates observed were invalid due to expiration, incorrect host names, or other reasons. Akhawe et al. [16] analyzed SSL certificates by monitoring live user traffic at several institutional networks, and provided a categorization of common certificate warnings, including server mis-configurations and browser design decisions. However, existing studies do not provide insights on forged certificates, probably since network attackers are relatively rare on those research institutional networks. In our work, we set out to measure real-world SSL connections from a large and diverse set of clients, in an attempt to find forged SSL certificates.

D. Tamper Detection Techniques for WebSites

Several techniques have been proposed to assist websites in detecting whether the client's network connections has been tampered with. In this paper, we focus on detection methods that do not require user interaction, and do not require

the installation of additional software or browser extensions. Notably, Web Tripwires [24] uses client-side JavaScript code to detect in-flight modifications to a web page. Several other studies [25], [26], [27], [28] have utilized Java applets to probe the client's network configurations and detect proxies that are altering the client's traffic.

- **Web Tripwires.** Web Tripwires [24] was a technique proposed to ensure data integrity of web pages, as an alternative to HTTPS. Websites can deploy JavaScript to the client's browser that detects modifications on web pages during transmission. In their study of real-world clients, over 1% of $50,000$ unique IP addresses observed altered web pages. Roughly 70% of the page modifications were caused by user-installed software that injected unwanted JavaScript into web pages. They found that some ISPs and enterprise firewalls were also injecting ads into web pages, or benignly adding compression to the traffic. Interestingly, they spotted three instances of client-side malware that modified their web pages. Web Tripwires was mainly designed to detect modifications to unencrypted web traffic. By design, Web Tripwires does not detect passive eavesdropping (that does not modify any page content), nor does it detect SSL man-in-the-middle attacks. In comparison, our goal is to be able to detect eavesdropping on encrypted SSL connections.
- **Content Security Policy.** Content Security Policy (CSP) [29] enables websites to restrict browsers to load page content, like scripts and stylesheets, only from a server-specified list of trusted sources. In addition, websites can instruct browsers to report CSP violations back to the server with the `report-uri` directive. Interestingly, CSP may detect untrusted scripts that are injected into the protected page, and report them to websites. Like Web Tripwires, CSP does not detect eavesdropping on SSL connections.
- **Browser Plugins.** Another technique for websites to diagnose the client's network is by using browser plugins, such as Java and Flash Player. Browser plugins may provide more network capabilities than JavaScript, including the ability to open raw network sockets and even perform DNS requests. For instance, the Illuminati [25] project used Java applets to identify whether clients were connecting through proxies or NAT devices. Jackson et al. conducted studies using both Java and Flash Player on real-world clients to find web proxy vulnerabilities, including multi-pin DNS rebinding [26] and cache poisoning [27]. The ICSI Netalyzer [28] used a signed Java applet to perform extensive tests on the client's network connectivity, such as detecting DNS manipulations.

In our work, we focused on detecting SSL man-in-the-middle attacks in real-world, from a website's perspective, without modifications to current browsers. Other proposals to prevent or mitigate SSL interception will be later discussed in Section V.

III. SSL Tamper Detection Method

In Section II-D, we discussed a number of existing techniques for websites to detect network tampering. However, none of the current methods (without browser modifications) are effective in detecting SSL man-in-the-middle attacks. In this section, we present a new method for detecting SSL man-in-the-middle attacks from the website's end. First, we describe our threat model. We then detail the design and our implementation of the detection method on the Facebook website. Lastly, we present our findings from analyzing millions of real-world SSL connections.

A. Threat Model

We primarily consider an *active network attacker* who has control over the victim's network connection. However, the attacker does not have control over the website (such as accessing internal machines and stealing the server's private key). The goal of the adversary is to read encrypted messages between the victim client and the HTTPS website. The attacker may impersonate the legitimate website with either (1) a trusted certificate issued by a trusted CA, or (2) an untrusted certificate (e.g. a self-signed certificate). In the case of an untrusted certificate, we assume that users may still be vulnerable, since previous studies [11], [12] have shown that many users ignore browser security warnings. Users are assumed to use up-to-date browsers (with no SSL implementation bugs).

In addition, we will discuss separately another type of *local attacker*, where the attacker may be a piece of software running on the client with the ability to modify the client's trusted CA store, as well as manipulate network connections. Such local attackers are much stronger than active network attackers, and are naturally not in scope of the SSL protocol's protection.

B. Design

There are several obstacles for websites to detect whether any SSL man-in-the-middle attacks are mounted against their connections. First of all, since SSL client certificates are rarely sent by normal users, it is not possible to distinguish a legitimate client from an attacker directly via the SSL handshake from the server's perspective. In order to determine whether an SSL connection is being intercepted, our fundamental approach is to observe the server's certificate from the client's perspective. Intuitively, if the client actually received a server certificate that does not exactly match the website's legitimate certificate, we would have direct evidence that the client's connection must have been tampered with.

Although one could easily develop a binary executable file or a custom browser extension that probes SSL certificates as an SSL client, it would not be scalable to distribute additional software to a large number of normal users, especially for non-tech-savvy users. Ideally, we would like to develop a JavaScript code snippet to observe SSL certificates, which runs in existing browsers and can reach a large population of clients. However, there are currently no existing browser APIs that allows web applications to directly check the observed

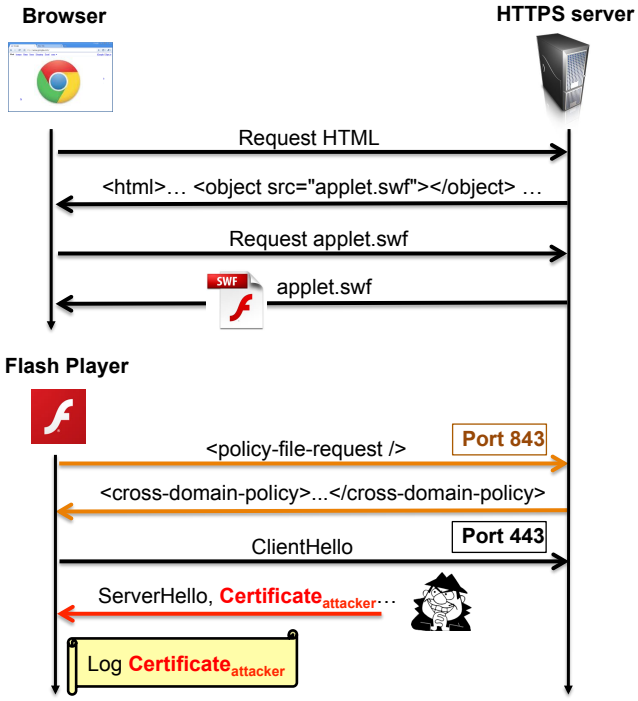

Fig. 3. The website loads a client-side applet, that performs the SSL handshake over a Flash-based socket connection to observe SSL certificates.

server certificate or validation status of their SSL connections. To workaround this, we utilized browser plugins to implement a client-side applet that is capable of imitating the browser's SSL handshake, accompanied with the ability to report the observed certificate chain. The applet can open a socket connection to the HTTPS server (skipping the browser's network stack), perform an SSL handshake over the socket, record the SSL handshake, and report the certificate chain back to our logging servers, shown in Figure 3. We describe our implementation details below.

1) Client-Side Applet: Our approach is to use a client-side applet that observes the server's SSL certificate from the client's perspective, directly during the SSL handshake. Since native browser networking APIs like XMLHttpRequest and WebSockets do not provide web applications access to raw bytes of socket connections, we must utilize browser plugins. We implemented a Shockwave Flash (SWF) applet that can open a raw socket connection to its own HTTPS server (typically on port 443), and perform an SSL handshake over the connection in the Flash Player.

By default, the Flash Player plugin does not allow any applets to access socket connections, unless the remote host runs a Flash socket policy server [30]. The Flash socket policy server, normally running on port 843, serves a socket policy file that declares whether SWF applications may open socket connections to the server. Note that even if a SWF file is requesting a socket connection to the same host it was served from, a socket policy server is still required. As a result, in order for a SWF applet from `example.com` to open a socket connection to a HTTPS server `example.com` on port 443, a valid socket policy file must be served at `example.com` on port 843, which permits socket access from `example.com` applications to port 443, as follows (in XML format):

```
<?xml version="1.0"?>
<!DOCTYPE cross-domain-policy SYSTEM
 "http://www.adobe.com/xml/dtds/cross-domain-policy.dtd">
<cross-domain-policy>
 <allow-access-from domain="example.com" to-ports="443" />
</cross-domain-policy>
```

Note that the socket policy file should not be confused with the `crossdomain.xml` file served by web servers, which restricts access to HTTP, HTTPS, and FTP access, but not socket access. If the Flash Player cannot successfully retrieve a valid socket policy (e.g. blocked by a firewall), the socket connection will be aborted and an exception will be thrown.

Once the socket connection is permitted, our applet will initiate an SSL handshake by sending a ClientHello message over the socket, and wait for the server to respond with the ServerHello and Certificate messages, which will be recorded. To support clients behind explicit HTTP proxies, the applet may send a CONNECT request over the socket to create an SSL tunnel prior to sending the ClientHello message, as follows:

```
CONNECT example.com:443 HTTP/1.1
```

Our SSL handshake implementation was based on the SSL 3.0 protocol version. Since our goal to observe the server's certificate chain, our applet closes the socket connection after successfully receiving the certificate chain. Lastly, our applet converts the raw bytes of the recorded SSL handshake responses into an encoded string, and sends it back to our log server with a POST request.

We note that the Flash Player plugin is currently supported on 95% of web browsers [31], therefore, our applet should be able to run on most clients. In fact, one of the major browsers, Google Chrome, has the Flash Player plugin built in by default. Also, SWF applets are usually allowed to execute without any additional user confirmation, and do not trigger any visual indicators (e.g. system tray icons) while running, thus, deploying this method should not affect the visual appearance of the original web page.

Alternatively, the client-side applet may be implemented using other browser plugins, for example, the Java plugin. Java applets are allowed to create socket connections from the client to any port on the same host that the applet was served from. As an example, an applet served from port 80 on `example.com` can open a raw socket to port 443 on `example.com` without requesting any additional access. However, due to security concerns, the Java plugin is currently blocked by default on several client platforms, and may require additional user interaction to activate the Java plugin. Such user interaction would be too obtrusive for our experiment and client diversity suffers greatly once all potential interactive platforms are removed from the experiment. Another side effect of running a Java applet on some platforms is that a

visible icon would be displayed in the system tray, which might annoy or confuse some of the website's users.

2) Lenient Certificate Extraction: Since we implemented the SSL handshake process on our own, we must extract the SSL certificates from a raw byte dump of the SSL handshake observed on the client, by parsing the ServerHello and ServerCertificate messages. Surprisingly, in our initial attempts, we found that this seemingly straightforward extraction process failed occasionally. By manual inspection, we noticed that some of the recorded SSL messages were slightly different from the SSL/TLS standards. As a result, we intentionally parsed the SSL handshake in as lenient a manner as possible in order to extract certificates even if the SSL message format did not conform exactly to the standards. We did not discard these malformed handshakes as we theorize that they are caused by either transmission errors or software errors in the intercepting proxy.

Websites may choose to perform certificate extraction on-the-fly in the client-side applet, or simply send the handshake raw bytes to their log servers for post-processing. We took the latter approach, since it enabled us to preserve the SSL handshake bytes for further investigation, even if early versions of our extraction code failed (or even crashed unexpectedly) while parsing certificates.

C. Implementation

We have implemented our client-side applets for both the Flash Player and Java plugins. With similar functionality, the SWF file (2.1 KB) was slightly smaller than the Java applet (2.5 KB). Since Flash Player was supported on a larger client population and is considered less obtrusive to users, we deployed the SWF file for our experiments.

To observe SSL connections on a large set of real-world clients, we deployed our client-side applet on Facebook's servers to run our experiments. We sampled a small portion (much less than 1%) of the total connections on Facebook's desktop website, particularly on the `www.facebook.com` domain. To avoid affecting the loading time of the website's original web pages, our applets are programmed to load several seconds after the original page has completed loading. This is done by using a JavaScript snippet that dynamically inserts a HTML object tag that loads the SWF file into the web page visited by the user. Basically, the script is triggered only after the original page finishes loading, and further waits a few seconds, before actually inserting the applet. Additionally, we built server-side mechanisms to allow granular control over the sampling rates in specific countries or networks. This granularity enables us to increase the sampling rate for certain populations in response to the detection of a specific attack.

To support Flash-based socket connections used by our SWF files, we have set up Flash socket policy servers that listens on port 843 of the website, which are configured with a socket policy file that allows only its own applets to open socket connections to port 443. We also setup a logging endpoint on the HTTPS servers, in PHP, that parses the reports, and aggregates data into our back-end databases. The extracted SSL certificates were processed and read using the OpenSSL library. In addition, we built an internal web interface for querying the log reports.

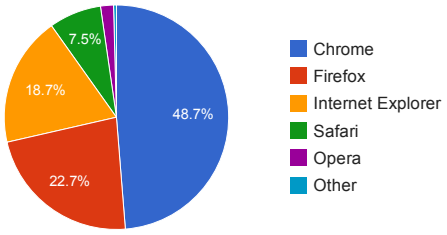

Fig. 4. Browser usage share of sampled clients. Note that given our sampling parameters, this is not directly representative of the entire population of Facebook's website.

TABLE I
NUMBER OF CLIENTS THAT COMPLETED EACH STEP OF THE DETECTION PROCEDURE

Procedure	Count
1. Inserted HTML object tag into web page	$9,179,453$
2. Downloaded SWF file from server	$6,908,675$
3. Sent report to logging server	$5,415,689$

D. Experimentation

Using the Flash-based detection method, we conducted the first large-scale experiment in an attempt to catch forged SSL certificates in the wild. We served our client-side applet to a set of randomly sampled clients on Facebook's website. We collected and analyzed data from November 20, 2012 to March 31, 2013.[2] Our dataset consists of reports from a variety of browsers, shown in Figure 4. The most popular browser versions in our dataset were (in descending order) Chrome 23, Chrome 24, Internet Explorer 9, Chrome 25, and Firefox 18.

First of all, we noticed that only a portion of the sampled clients actually completed our detection procedure, explained below. As shown in Table I, a total of $9,179,453$ page views on Facebook's desktop website had our HTML object tag dynamically inserted. Our web servers logged $6,908,675$ actual downloads for the SWF file. The download count for the SWF file was noticeably lower than the number of object tags inserted. We reason that this is possibly due to: (1) the Flash Player plugin was not enabled on the client, (2) a few legacy browsers did not support our SWF object embedding method, or (3) the user navigated away from the web page before the object tag was loaded. Our log servers received a total of $5,415,689$ reports from applets upon successful execution. Again, the number of received reports is lower than the number of SWF file downloads. This is likely due to the web page being closed or navigated away by the user, before the applet was able to finish execution.

[2]Personally identifiable information (IP addresses and HTTP cookies) were removed from our database after a 90-day retention period.

TABLE II
CATEGORIZATION OF REPORTS

Type	Count
Well-formed certificates	3,447,719 (64%)
Flash socket errors	1,965,186 (36%)
Empty reports	2,398 (0%)
Bogus reports	290 (0%)
HTTP responses	96 (0%)

Next, we noticed that only 64% out of the 5,415,689 received reports contained complete and well-formed certificate records, as shown in Table II. We observed that 1,965,186 (36%) of the reported data indicated that the client caught SecurityErrorEvent or IOErrorEvent exceptions in the Flash Player and failed to open a raw socket. We believe that most of these errors were caused by firewalls blocking the socket policy request (for example, whitelisting TCP ports 80 and 443 to only allow web traffic), thus not allowing the Flash Player to retrieve a valid socket policy file from our socket policy servers (over port 843). For clients behind these firewalls, we were not able to open socket connections using Flash Player, although using Java might have worked in some legacy client platforms. We discuss in Section III-E that similar measurements can be conducted on native mobile platforms to avoid the drawbacks of Flash sockets.

In addition to the Flash socket errors, we also observed a few other types of erroneous reports. There were 2,398 reports that were empty, indicating that the SWF file failed to receive any certificates during the SSL handshake. This might have been caused by firewalls that blocked SSL traffic (port 443). There were 96 reports that received HTTP responses during the SSL handshake, mostly consisting of error pages (HTTP `400` code) or redirection pages (HTTP `302` code). These responses suggest that some intercepting proxies contained logic that were modifying the client's web traffic to block access to certain websites (or force redirection to certain web pages, known as captive portals). We found that some clients received a HTML page in plaintext over port 443, for instance, linking to the payment center of Smart Bro, a Philippine wireless service provider. These type of proxies do not appear to eavesdrop SSL traffic, but they inject unencrypted HTTP responses into the client's web traffic.

In addition, there were 290 reports that contained garbled bytes that could not be correctly parsed by our scripts. Although we could not successfully parse these reports, manual inspection determined that 16 of the reports contained seemingly legitimate VeriSign certificates that had been truncated in transit, presumably due to lost network connectivity. Another 37 of these reports appear to be issued by Kurupira.NET, a web filter, which closed our SSL connections prematurely. We also found that 17 of the unrecognized POST requests on our log servers were sent from a Chrome extension called Tapatalk Notifier (determined by the HTTP `origin` header), however we have no evidence that these false POST requests were intentional.

Finally, we successfully extracted 3,447,719 (64%) well-formed certificates from the logged reports. We used custom scripts (mentioned in Section III-B2) to parse the recorded SSL handshake bytes. A total of 3,440,874 (99.8%) out of 3,447,719 observed certificates were confirmed to be the website's legitimate SSL certificates, by checking the RSA public keys (or more strictly, by comparing the observed certificate bit-by-bit with its legitimate certificates). We note that there were multiple SSL certificates (thus, multiple RSA public keys) legitimately used by Facebook's SSL servers during the period of our study, issued by publicly-trusted commercial CAs including VeriSign, DigiCert, and Equifax. Most interestingly, we discovered that 6,845 (0.2%) of the observed certificates were not legitimate, nor were they in any way approved by Facebook. We further examine these captured forged certificates in Section IV.

E. Limitations

Before we move on, we offer insights on the limitations of our detection method. It is important to point out that the goal of our implementation was not to evade the SSL man-in-the-middle attacks with our detection mechanism. Admittedly, it would be difficult to prevent professional attackers that are fully aware of our detection method. We list below some ways that an attacker might adaptively evade our detection:

- Attackers may corrupt all SWF files in transmission, to prevent our client-side applet from loading. However, this approach would cause many legitimate applications using SWF files to break. Of course, the attacker could narrow the scope of SWF blacklisting to include only the specific SWF files used in this detection. In response, websites may consider randomizing the locations of their SWF files.
- Attackers may restrict Flash-based sockets by blocking Flash socket policy traffic on port 843. To counter this, websites could possibly serve socket policy files over firewall-friendly ports (80 or 443), by multiplexing web traffic and socket policy requests on their servers. In addition, websites could try falling back to Java applets on supporting clients if Flash-based sockets are blocked.
- Attackers may try to avoid intercepting SSL connections made by the Flash Player. However, the website may tailor its client-side applet to act similarly to a standard browser.
- In theory, attackers could possibly tamper the reports (assuming that the measured client was under an SSL man-in-the-middle attack, and probably clicked through SSL warnings, if any), and trick our log servers to believe that the website's legitimate certificate was observed. Under this scenario, the website may need additional mechanisms to verify the integrity of their reports.

At the time of this study, there is no reason to think that any attacker is tampering our reports, or even aware of our detection method. We do not consider attackers that have

TABLE III
FORGED CERTIFICATE CHAIN SIZES

Size (bytes)	Count
0 - 1000	6,154 (90%)
1000 - 2000	508 (7%)
2000 - 3000	140 (2%)
3000 - 4000	29 (0%)
4000 - 5000	2 (0%)
5000 - 6000	23 (0%)
6000 - 7000	13 (0%)

obtained access to Facebook's internal servers. As shown in Section III-D, our current methodology has successfully captured direct evidences of unauthorized SSL interceptions in the wild. However, if more websites become more aggressive about this sort of monitoring, we might get into an arms race, unfortunately.

Fortunately, many popular websites nowadays have the option to leverage their native mobile applications for detecting attacks. While our initial implementation targeted desktop browsers, we suggest that similar mechanisms can be implemented, more robustly, on mobile platforms such as iOS and Android.[3] Native mobile applications have the advantage of opening socket connections without Flash-based socket policy checks, and are more difficult for network attackers to bypass (since the Flash applet is no longer necessary, and native applications can be programmed to act exactly like a standard browser). Furthermore, mobile clients can also implement additional defenses (e.g. certificate pinning [22]) to harden itself against SSL man-in-the-middle attacks (e.g. preventing the tampering of reports), while performing similar measurement experiments.

IV. ANALYSIS OF FORGED SSL CERTIFICATES

From the experiments in Section III-D, we collected $6,845$ forged certificates from real-world clients connecting to Facebook's SSL servers. In this section, we analyze the root cause of these injected forged SSL certificates. First, we survey the characteristics of the forged certificate chains, including the certificate chain sizes, certificate chain depths, and public key sizes. Subsequently, we examine the subject names and the issuer names of the forged certificates.

A. Size Characteristics

We first examine the size characteristics of the forged SSL certificates, as follows:

- **Certificate chain sizes.** Table III summarizes the total sizes in bytes of the forged certificate chains. Notably, most of the forged certificate chains were actually very small (less than a kilobyte). By manual inspection, these small certificates were generally self-signed certificates that did not include any intermediate CA certificates (thus the smaller chain size). A small number of certificate chains were larger than 5 KB in size, where the size overhead might have a negative impact on page load time for victim users.

[3]After our initial study, Facebook has implemented our methodology across their native mobile applications.

TABLE IV
FORGED CERTIFICATE CHAIN DEPTHS

Depth	Count
1	6,173 (90%)
2	617 (9%)
3	19 (0%)
4	34 (0%)
5	2 (0%)

TABLE V
PUBLIC KEY SIZES OF FORGED SERVER CERTIFICATES

Public Key Size (bits)	Count
512	119 (2%)
1024	3,447 (50%)
2048	3,279 (48%)

- **Certificate chain depths.** Table IV shows the distribution of certificate chain length. Here, we refer to the certificate chain depth as the number of certificates (including any intermediate CA certificates) actually transmitted during the SSL handshake. We note that on most websites, the certificate chains normally do not include the issuing root CA certificate (since trusted CA certificates are presumed to be installed on the client, thus omitted in transmission). The majority of the forged certificate chains have a depth of one, which only contained the server's end entity certificate without any intermediate certificates. Since most commercial CAs nowadays issue certificates using intermediate keys (rather than their root keys), one should probably raise some suspicion when encountering certificate chains with a depth of one. There were 55 of the forged certificates that had a depth of 3 or larger, which is actually longer than the website's legitimate certificate chain. For these certificate chains, additional cryptographic computations or even online revocation checks on the client might be required, since the client needs to verify signatures for all of the intermediate certificates when establishing an SSL connection. On slower devices, the additional verification time might be noticeable by the victim user.
- **Public key sizes.** Table V shows the RSA public key sizes in bits carried in the forged certificates. Most of the forged certificates had either 1024-bit or 2048-bit public keys, which are not characteristically different from legitimate SSL certificates (although websites should transition to 2048-bit or stronger RSA keys by 2014 according to the CA/Browser forum's recommendations). We noticed that a few certificates actually contained relatively weak 512-

TABLE VI
SUBJECT ORGANIZATIONS OF FORGED CERTIFICATES

Subject Organization	Count
Facebook, Inc.	6,552
Empty	131
Fortinet Ltd. / Fortinet	93
Lousville Free Public Library	10
Other	59

TABLE VII
SUBJECT COMMON NAMES OF FORGED CERTIFICATES

Subject Common Name	Count
`*.facebook.com`	6,491
`www.facebook.com`	117
`pixel.facebook.com`	1
`m.facebook.com`	1
`facebook.com`	1
`*`	1
IP addresses	118
`FG...` / Fortinet / FortiClient	93
Other	22

bit public keys. These users may have become further vulnerable to a *second* attacker given the considerably weakened public key.

B. Certificate Subjects

First, Table VI shows the subject organizations of forged certificates. As expected, the majority of them spoofed the organization as Facebook. There were over a hundred forged certificates that excluded the organization attribute entirely. Again, we confirmed 93 certificates that were attributed to Fortinet Ltd.

Next, we inspect the observed subject common names of the forged SSL certificates, summarized in Table VII. Normally, the subject common name of the SSL certificate should match the hostname of the website to avoid triggering SSL certificate warnings in the browser. While most of the forged certificates used the legitimate website's domains as the subject common name, there were a few certificates that used unrelated domains as well.

Unsurprisingly, most of the forged SSL certificates used the wildcard domain `*.facebook.com` as the subject common name in order to avoid certificate name validation errors. This suggests that most of the attacking entities were either specifically targeting Facebook's website by pre-generating certificates that match the website's name, or using automated tools to generate the certificates on-the-fly. None of the forged certificates were straight clones of Facebook's legitimate certificates (that replicated all the X.509 extension fields and values). There were some certificates that used IP addresses as common name, for example, `69.171.255.255` (which appears to be one of Facebook's server IP addresses). We noticed that a number of forged certificates used a subject name that starts with two characters `FG` concatenated with a long numeric string (e.g. `FG600B3909600500`). These certificates were issued by Fortinet Ltd., a company that manufactures SSL proxy devices which offer man-in-the-middle SSL inspection. Similarly, we found 8 certificates that had a subject common name "labris.security.gateway SSL Filtering Proxy," which is also an SSL proxy device. There were a few other common names observed that were likely amateur attempts of SSL interception, such as `localhost.localdomain`, which is the default common name when generating a self-signed certificate using the OpenSSL library.

For the forged SSL certificates that did not use a subject common name with `facebook.com` as suffix, we also checked if any subject alternative names were present in the certificate. Subject alternative names are treated as additional subject names, and allow certificates to be shared across multiple distinct hostnames. This may allow attackers to generate a single forged certificate for attacking multiple different websites. For the 233 forged certificates that did not provide a matching common name, none of them provided a matching subject alternative name. Even though these 233 (3.4%) forged certificates would definitely trigger name mismatch errors, there is still a significant possibility that users may ignore the browser's security warnings anyway.

C. Certificate Issuers

In this section, we examine the issuer organizations and issuer common names of each forged SSL certificate. Table VIII lists the top issuer organizations of the forged certificates. At first glance, we noticed several forged certificates that fraudulently specified legitimate organizations as the issuer, including 5 using Facebook, 4 using Thawte, and one using VeriSign. These invalid certificates were not actually issued by the legitimate companies or CAs, and were clearly malicious attempts of SSL interception. Since 166 of the forged certificates did not specify its issuer organization (or *empty*), we also checked the issuer common names, listed in Table IX.

We manually categorized the certificate issuers of forged certificates into antivirus, firewalls, parental control software, adware, and malware. Notably, we observed an intriguing issuer named lopFailZeroAccessCreate that turned out to be produced by malware, which we discuss in detail below.

- **Antivirus.** By far the top occurring issuer was Bitdefender with $2,682$ certificates, an antivirus software product which featured a "Scan SSL" option for decrypting SSL traffic. According to their product description, Bitdefender scans SSL traffic for the presence of malware, phishing, and spam. The second most common issuer was ESET with $1,722$ certificates, another antivirus software product that provides SSL decryption capabilities for similar purposes. Several other top issuers were also vendors of antivirus software, such as BullGuard, Kaspersky Lab, Nordnet, DefenderPro, etc. These software could possibly avoid triggering the browser's security errors by installing their self-signed root certificates into the client's

TABLE VIII
ISSUER ORGANIZATIONS OF FORGED CERTIFICATES

Issuer Organization	Count
Bitdefender	2,682
ESET, spol. s r. o.	1,722
BullGuard Ltd.	819
Kaspersky Lab ZAO / Kaspersky Lab	415
Sendori, Inc	330
Empty	166
Fortinet Ltd. / Fortinet	98
EasyTech	78
NetSpark	55
Elitecore	50
ContentWatch, Inc	48
Kurupira.NET	36
Netbox Blue / Netbox Blue Pty Ltd	25
Qustodio	21
Nordnet	20
Target Corporation	18
DefenderPro	16
ParentsOnPatrol	14
Central Montcalm Public Schools	13
TVA	11
Louisville Free Public Library	10
Facebook, Inc.	5
thawte, Inc.	4
Oneida Nation / Oneida Tribe of WI	2
VeriSign Trust Network	1
Other (104)	186

TABLE IX
ISSUER COMMON NAMES OF FORGED CERTIFICATES

Issuer Common Name	Count
Bitdefender Personal CA.Net-Defender	2,670
ESET SSL Filter CA	1,715
BullGuard SSL Proxy CA	819
Kaspersky Anti-Virus Personal Root Certificate	392
Sendori, Inc	330
IopFailZeroAccessCreate	112
...	
`*.facebook.com`	6
VeriSign Class 4 Public Primary CA	5
Production Security Services	3
Facebook	1
thawte Extended Validation SSL CA	1
Other (252)	794

system. Note that the observed antivirus-related certificate counts are not representative of the general antivirus usage share of the website's users, since SSL interception is often an optional feature in these products. However, if any antivirus software enabled SSL interception by default, we would expect a higher number of their forged certificates observed.

Supposing that these users intentionally installed the antivirus software on their hosts, and deliberately turned on SSL scanning, then these antivirus-generated certificates would be less alarming. However, one should be wary of professional attackers that might be capable of stealing the private key of the signing certificate from antivirus vendors, which may essentially allow them to spy on the antivirus' users (since the antivirus' root certificate would be trusted by the client). Hypothetically, governments could also compel antivirus vendors to hand over their signing keys.

- **Firewalls.** The second most popular category of forged certificates belongs to commercial network security appliances that perform web content filtering or virus scanning on SSL traffic. As observed in the certificate subject fields, Fortinet was one of the issuers that manufactures devices for web content filtering with support for HTTPS deep inspection. NetSpark was another web content filtering device manufacturer offering similar capabilities. According to their product description, the user's content is unencrypted for inspection on NetSpark's servers, and then re-encrypted under NetSpark's SSL certificate for the end user. We observed a number of device vendors that provided similar devices, such as EliteCore, ContentWatch, and Netbox Blue. There were also software solutions that provided selective website blocking, such as Kurupira.NET. Some appliance vendors aggressively marketed SSL content inspection as a feature which cannot be bypassed by users. For example, ContentWatch's website provided the following product description for their firewall devices:[4]

  > *"This technology also ensures the users cannot bypass the filtering using encrypted web traffic, remote proxy servers or many of the other common methods used circumvent content filters."*

  Interestingly, EliteCore's Cyberoam appliances have previously been discovered [32] to be using the same CA private key across all Cyberoam devices. This is particularly dangerous, since the universal CA private key can be extracted from any single device by an attacker. This vulnerabilitiy allows an attacker to seamlessly perform SSL man-in-the-middle attacks against users of benign Cyberoam devices, because the attacker can issue forged server certificates that will be accepted by other clients that have installed Cyberoam's CA certificate. Reportedly, Cyberoam issued an over-the-air patch to generate unique CA certificates on each device. Nevertheless, we should be aware that other device manufacturers are likely to introduce similar security vulnerabilities.
- **Adware.** We observed 330 instances of forged certificates

[4]http://www.contentwatch.com/solutions/industry/government

issued by a company named Sendori. This company offers a browser add-on that claims to automatically correct misspelled web pages. However, using Google Search to query the string "Sendori" revealed alarming discussions about the add-on actually hijacking DNS entries for the purposes of inserting advertisements into unrelated websites.[5] This form of adware actively injects content into webpages, and could possibly be detected using Web Tripwires or CSP (as described in Section II-D).

- **Malware.** As previously mentioned, we noticed that an unknown issuer named lopFailZeroAccessCreate appeared relatively frequently in our dataset. We manually searched the name on the Internet and noticed that multiple users were seeing SSL certificate errors of the same issuer, and some were suggesting that the user could be under SSL man-in-the-middle attacks by malware.[6] Upon deeper investigation, we discovered 5 forged certificates that shared the same subject public key as lopFailZeroAccessCreate, yet were generated with their issuer attribute set as "VeriSign Class 4 Public Primary CA." We confirmed with Symantec/VeriSign that these suspicious certificates were not issued through their signing keys. This was obviously a malicious attempt to create a certificate with an issuer name of a trusted CA. These variants provide clear evidence that attackers in the wild are generating certificates with forged issuer attributes, and even increased their sophistication during the time frame of our study.

  In Figure 5, we illustrate the geographic distribution of the certificates issued by lopFailZeroAccessCreate (and the forged "VeriSign Class 4 Public Primary CA") on a world map. As shown, the infected clients were widespread across 45 different countries. The countries with the highest number of occurrences were Mexico, Argentina and the United States, with 18, 12, and 11 occurrences, respectively. This shows that the particular SSL man-in-the-middle attack is occurring globally in the wild. While it is possible that all of these attacks were amateur attackers individually mounting attacks (e.g. at their local coffee shop), it is certainly odd that they happened to use forged certificates with the same subject public key. However, this is not so unreasonable if these attacks were mounted by malware. Malware researchers at Facebook, in collaboration with the Microsoft Security Essentials team, were able to confirm these suspicions and identify the specific malware family responsible for this attack. Since our experiments only tested SSL connections to Facebook's servers (only for the `www.facebook.com` domain), we cannot confirm whether this attack also targeted other websites. In response to our discovery, the website notified the infected users, and provided them with malware scan and repair instructions.

[5]http://helpdesk.nwciowa.edu/index.php?/News/NewsItem/View/10
[6]http://superuser.com/q/421224

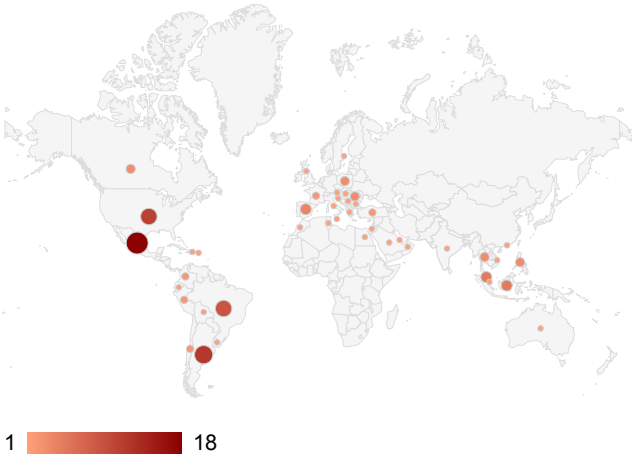

Fig. 5. Geographic distribution of forged SSL certificates generated by the malicious issuer lopFailZeroAccessCreate

  In addition, there were 4 other suspicious certificates issued under the organization name of thawte, Inc with three of them using "Production Security Services" as the issuer common name, and one using "thawte Extended Validation SSL CA." These instances could be the same malware attack previously spotted by some Opera users [33], in which forged certificates pretending to be issued by Thwate were observed. These 4 forged certificates were observed in Italy, Spain, and the United States.

  We note that a sophisticated attacker utilizing malware could install their self-signed CA certificates on clients in order to suppress browser security errors. Such an attacker is likely capable of stealing confidential information, by reading from protected storage or logging the user's keystrokes. Nevertheless, mounting an SSL man-in-the-middle attack further enables a general way of capturing and recording the victim's web browsing activities in real-time.

- **Parental Control Software.** Some forged SSL certificates were issued by parental control software, including 21 from Qustodio and 14 from ParentsOnPatrol. These type of software are designed to enable parents to monitor and filter the online activities of their children. Whether such level of control is appropriate is beyond the scope of our work.

While the remaining 104 other distinct issuer organizations in Table VIII and 252 other distinct common names in Table IX do not appear to be widespread malicious attempts (based on manual inspection), the possibility remains that some may still be actual attacks.

For example, we found two unexpected instances of forged certificates issued by the Oneida Nation of Wisconsin, an Indian tribe. We have little clue of why encrypted traffic would be eavesdropped by such entities. It is possible that this is another case of corporate surveillance. We also found that

some schools and libraries were using forged certificates for SSL interception, such as "Central Montcalm Public Schools" and "Louisville Free Public Library."

For some certificates, the certificate attributes alone provide insufficient clues to determine their origin. For example, an issuer named EasyTech could either implicate the PC repair service at Staples EasyTech, or the EasyTech digital education system on `learning.com`. In either case, we were unclear why SSL connections were being intercepted for those particular targets.

V. Survey of Mitigations

We provided direct evidences of a variety of forged SSL certificates from real-world connections in Section IV. In this section, we list some possible defenses that websites or browser vendors may consider to help mitigate these attacks.

A. Strict Transport Security

HTTP Strict Transport Security (HSTS) [34], the successor of ForceHTTPS [35], is a HTTP response header that allows websites to instruct browsers to make SSL connections mandatory on their site. By setting the HSTS header, websites may prevent network attackers from performing SSL stripping [36]. A less obvious security benefit of HSTS is that browsers simply hard-fail when seeing invalid certificates, and do not give users the option to ignore SSL errors. This feature prevents users from accepting untrusted certificates when under man-in-the-middle attacks by amateur script kiddies. However, HSTS is not designed to protect against malware or professional attackers that use forged certificates that would be accepted by the browser.

B. Public Key Pinning

The Public Key Pinning Extension for HTTP (HPKP) [37] proposal allows websites to specify their own public keys with a HTTP header, and instruct browsers to reject any certificates with unknown public keys. HPKP provides protection against SSL man-in-the-middle attacks that use unauthorized, but possibly trusted, certificates. HPKP automatically rejects fraudulent certificates even if they would be otherwise trusted by the client. Both HSTS and HPKP defenses require that clients must first visit the legitimate website securely before connecting from untrusted networks. This requirement is lifted if public key pins are pre-loaded in the browser, such as in Google Chrome [38] and Internet Explorer (with EMET) [39], although this approach may not scale for the entire web. Notably, Chrome's pre-loaded public key pinning mechanism has successfully revealed several high-profile CA incidents, in which mis-issued certificates were used to attack Google's domains in the wild. However, in current implementations, Chrome's public key pinning does not reject certificates that are issued by a locally trusted signer, such as antivirus, corporate surveillance, and malware.

A related proposal, Trust Assertions for Certificate Keys (TACK) [40], allows SSL servers to pin a server-chosen signing key with a TLS extension. In contrast with HPKP, TACK pins a signing key that chosen by the server, separate from the private key corresponding to the server's certificate, and can be short-lived. TACK allows websites with multiple SSL servers and multiple public keys to pin the same signing key. Once the browser receives a TACK extension from an SSL site, it will require future connections to the same site to be signed with the same TACK signing key, or otherwise, reject the connection. Another proposal called DVCert [41] delivers a list of certificate pins over a modified PAKE protocol in an attempt to detect SSL man-in-the-middle attacks, but also requires modifications to existing clients and servers.

The concept of public key pinning (or certificate pinning) has previously been implemented as a pure client-side defense as well. Firefox add-ons such as Certificate Patrol [42] and Certlock [9] were designed to alarm users when a previously visited website starts using a different certificate. However, without explicit signals from the server, it may be difficult to accurately distinguish real attacks from legitimate certificate changes, or alternative certificates.

C. Origin-Bound Certificates

The TLS Origin-Bound Certificates (TLS-OBC) [43] proposal revisits TLS client authentication, by enabling browsers to generate self-signed client certificates on demand without requiring any user configurations. TLS-OBC may block most of the existing man-in-the-middle attack toolkits, since attackers cannot impersonate the client (without stealing the self-signed private key from the legitimate browser). However, it does not prevent an impersonated server from supplying a cacheable malicious JavaScript file to the client, which later executes in the context of the victim website, and potentially exfiltrates data by reconnecting to the legitimate server. Further, TLS-OBC requires code changes to the network stack on servers (while HSTS and HPKP do not), and induces extra computational costs for client certificate generation. Websites should assess whether this is an acceptable trade-off.

D. Certificate Validation with Notaries

Perspectives [44] is a Firefox add-on that compares server certificates against multiple notaries (with different network vantage points) to reveal inconsistencies. Since public notaries observe certificates from diverse network perspectives, a local impersonation attack could be easily detected. Convergence [45] extends Perspectives by anonymizing the certificate queries for improved privacy, while allowing users to configure alternative verification methods (such as DNSSEC). The DetecTor [46] project (which extends Doublecheck [47]) makes use of the distributed Tor network to serve as external notaries. Crossbear [48] further attempts to localize the attacker's position in the network using notaries. However, notary approaches might produce false positives when servers switch between alternative certificates, and clients may experience slower SSL connection times due to querying multiple notaries during certificate validation. Further, these pure client-side defenses have not been adopted by mainstream browsers, thus cannot protect the majority of (less tech-savvy) users.

E. Certificate Audit Logs

Several proposals have suggested the idea of maintaining cryptographically irreversible records of all the legitimately-issued certificates, such that mis-issued certificates can be easily discovered, while off-the-record certificates are simply rejected. Sovereign Keys [49] requires clients to query public timeline servers to validate certificates. Certificate Transparency (CT) [50] removes the certificate queries from clients by bundling each certificate with an audit proof of its existence in the public log. Accountable Key Infrastructure (AKI) [51] further supports revocation of server and CA keys. These defenses are designed to protect against network attackers (not including malware). However, browsers need to be modified to support the mechanism, and changes (or cooperation) are needed on the CAs or servers to deliver the audit proof. Encouragingly, Google has announced their plan to use Certificate Transparency for all EV certificates in the Chrome browser [52].

F. DNS-based Authentication

DNS-based Authentication of Named Entities (DANE) [53] allows the domain operator to sign SSL certificates for websites on its domain. Similar to public key pinning defenses, DANE could allow websites to instruct browsers to only accept specific certificates. This approach prevents any CAs (gone rogue) from issuing trusted certificates for any domain on the Internet. Another related proposal, the Certification Authority Authorization (CAA) [54] DNS records, can specify that a website's certificates must be issued under a specific CA. However, these approaches fundamentally rely on DNSSEC to prevent forgery and modification of the DNS records. Until DNSSEC is more widely deployed on the Internet, websites must consider alternative defenses.

G. Discussion

In this last section, we analyze the robustness of the possible defenses with regards to four common types of SSL man-in-the-middle attackers:

1) **Script kiddie.** A script kiddie is a relatively unskilled individual who simply downloads attack toolkits that have been created by others. These attackers generally only attempt to perform attacks with less sophisticated methods (with self-signed forged certificates) and on a smaller scale, such as a local public WiFi hotspot. All of the defenses are immune to script kiddies. Even HSTS can block these attacks, because any invalid certificates will hard-fail in supporting browsers.
2) **Corporate-level surveillance.** Most SSL man-in-the-middle attacks are attributed to corporate-level surveillance. In such scenarios, the corporate IT technicians may access the client's machine to establish trust with their self-signed root certificates. SSL interception is typically done without any authorization from the legitimate websites. HSTS, by design, does not reject browser-accepted certificates. HPKP also does not reject certificates signed by locally trusted CAs, as mentioned in Section V-B. Other defenses including audit logs and notary-based approaches can detect these attacks.
3) **Professional attacker.** Next, we consider the professional attackers, which we define as an entity that has managed to obtained a forged certificate from a trusted CA. This classification may include a state-sponsored attacker who may compel trusted CAs to issue forged certificates, or a sophisticated hacker who has successfully compromised a trusted CA directly. Server-side defenses including CT, AKI, SK, TACK, and HPKP are designed to block this type of mis-issued certificates, although at varying deployment costs (e.g. HPKP is the most lightweight in terms of server modifications). Notary-based defenses may spot local inconsistencies if the attacker mounted the attacks discriminately.
4) **Malware.** As discussed in Section IV-C, malware nowadays may also perform SSL interception. None of the possible defenses are designed to prevent malware attackers, which may have access to the victim client's machine, and simply tamper with the client's root CA store.

In all, websites may consider reaping the security benefits of deploying HSTS and HPKP defenses in conjunction, since they are most readily supported in at least one of the major browsers today. Unfortunately, these two defenses do not prevent all types of attacks such as corporate-level surveillance, as discussed. Several defense proposals (and prototypes) are more robust against most attacks (with the exception of malware, which is out-of-scope), but are not yet available for websites to use. Websites may want to stay agile and deploy multiple approaches for defense in depth. In the meantime, we recommend that websites or mobile applications can adopt our detection method (as publicized in this paper), and possibly collaborate to detect SSL interceptions in the wild.

VI. Conclusion

In this paper, we introduced a new method for detecting SSL man-in-the-middle attacks against a website's users. We demonstrated the feasibility of detecting man-in-the-middle attacks by implementing this mechanism on millions of SSL connections at a top global website, Facebook. We presented the first analysis of forged SSL certificates in the wild. We revealed direct evidences that 0.2% of real-world connections were substituted with unauthorized forged certificates. While most of the SSL interceptions were due to antivirus software and corporate surveillance devices, we also observed a few amateur attack attempts, and even traces of pervasive malware in the wild that intercepted SSL connections. Our data suggest that browsers could possibly detect many of the forged certificates based on size characteristics, such as checking whether the certificate chain depth is larger than one. We strongly encourage popular websites, as well as mobile applications, to deploy similar mechanisms to start detecting SSL interception. Lastly, we assessed possible mitigations for SSL man-in-the-middle attacks, and recommend websites to deploy multiple available defenses, in conjunction, for better protection.

ACKNOWLEDGMENT

This work was done during Huang's internship at Facebook. We thank Adam Langley, Scott Renfro, Zack Weinberg, and the anonymous reviewers for providing feedback on drafts of the paper. Special thanks to Mark Hammell and Joren McReynolds for their code reviews.

REFERENCES

[1] A. O. Freier, P. Karlton, and P. C. Kocher, "The Secure Sockets Layer (SSL) Protocol Version 3.0," RFC 6101 (Historic), Internet Engineering Task Force, Aug. 2011.

[2] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008.

[3] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile," RFC 5280 (Proposed Standard), Internet Engineering Task Force, May 2008.

[4] Electronic Frontier Foundation, "The EFF SSL Observatory," https://www.eff.org/observatory.

[5] VASCO, "DigiNotar reports security incident," http://www.vasco.com/company/about_vasco/press_room/news_archive/2011/news_diginotar_reports_security_incident.aspx, Aug. 2011.

[6] Comodo, "Comodo Report of Incident - Comodo detected and thwarted an intrusion on 26-MAR-2011," http://www.comodo.com/Comodo-Fraud-Incident-2011-03-23.html, Mar. 2011.

[7] TURKTRUST, "Public announcements," http://turktrust.com.tr/en/kamuoyu-aciklamasi-en.html, Jan. 2013.

[8] H. Adkins, "An update on attempted man-in-the-middle attacks," http://googleonlinesecurity.blogspot.com/2011/08/update-on-attempted-man-in-middle.html.

[9] C. Soghoian and S. Stamm, "Certified lies: detecting and defeating government interception attacks against SSL," in *Proceedings of the 15th International Conference on Financial Cryptography and Data Security*, 2011.

[10] S. E. Schechter, R. Dhamija, A. Ozment, and I. Fischer, "The emperor's new security indicators," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[11] J. Sunshine, S. Egelman, H. Almuhimedi, N. Atri, and L. F. Cranor, "Crying wolf: an empirical study of SSL warning effectiveness," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[12] D. Akhawe and A. P. Felt, "Alice in warningland: A large-scale field study of browser security warning effectiveness," in *Proceedings of the 22nd USENIX Security Symposium*, 2013.

[13] A. P. Felt, H. Almuhimedi, S. Consolvo, and R. W. Reeder, "Experimenting at scale with Google Chrome's SSL warning," in *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2014.

[14] P. Eckersley, "A Syrian man-in-the-middle attack against Facebook," https://www.eff.org/deeplinks/2011/05/syrian-man-middle-against-facebook, May 2011.

[15] R. Holz, L. Braun, N. Kammenhuber, and G. Carle, "The SSL landscape: a thorough analysis of the x.509 PKI using active and passive measurements," in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, 2011.

[16] D. Akhawe, B. Amann, M. Vallentin, and R. Sommer, "Here's my cert, so trust me, maybe? Understanding TLS errors on the web," in *Proceedings of the International Conference on World Wide Web*, 2013.

[17] Z. Durumeric, J. Kasten, M. Bailey, and J. A. Halderman, "Analysis of the HTTPS certificate ecosystem," in *Proceedings of the 13th ACM SIGCOMM Conference on Internet Measurement*, 2013.

[18] E. Butler, "Firesheep," http://codebutler.com/firesheep.

[19] S. Chen, Z. Mao, Y.-M. Wang, and M. Zhang, "Pretty-Bad-Proxy: An overlooked adversary in browsers' HTTPS deployments," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009.

[20] 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 ACM Conference on Computer and Communications Security*, 2012.

[21] 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 ACM Conference on Computer and Communications Security*, 2012.

[22] S. Fahl, M. Harbach, H. Perl, M. Koetter, and M. Smith, "Rethinking SSL development in an appified world," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013.

[23] M. Marlinspike, "sslsniff," http://www.thoughtcrime.org/software/sslsniff.

[24] C. Reis, S. D. Gribble, T. Kohno, and N. C. Weaver, "Detecting in-flight page changes with web tripwires," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[25] M. Casado and M. J. Freedman, "Peering through the shroud: the effect of edge opacity on IP-based client identification," in *Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007.

[26] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh, "Protecting browsers from DNS rebinding attacks," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[27] L.-S. Huang, E. Y. Chen, A. Barth, E. Rescorla, and C. Jackson, "Talking to yourself for fun and profit," in *Proceedings of the Web 2.0 Security and Privacy*, 2011.

[28] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson, "Netalyzr: Illuminating the edge network," in *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, 2010.

[29] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the 19th International Conference on World Wide Web*, 2010.

[30] P. Uhley, "Setting up a socket policy file server," http://www.adobe.com/devnet/flashplayer/articles/socket_policy_files.html, Apr. 2008.

[31] StatOwl.com, "Flash player version market share and usage statistics," http://www.statowl.com/flash.php.

[32] R. A. Sandvik, "Security vulnerability found in Cyberoam DPI devices (CVE-2012-3372)," https://blog.torproject.org/blog/security-vulnerability-found-cyberoam-dpi-devices-cve-2012-3372, Jul. 2012.

[33] Y. N. Pettersen, "Suspected malware performs man-in-the-middle attack on secure connections," http://my.opera.com/securitygroup/blog/2012/05/16/suspected-malware-performs-man-in-the-middle-attack-on-secure-connections, May 2012.

[34] J. Hodges, C. Jackson, and A. Barth, "HTTP Strict Transport Security (HSTS)," RFC 6797 (Proposed Standard), Internet Engineering Task Force, Nov. 2012.

[35] C. Jackson and A. Barth, "ForceHTTPS: protecting high-security web sites from network attacks," in *Proceedings of the 17th International Conference on World Wide Web*, 2008.

[36] M. Marlinspike, "New techniques for defeating SSL/TLS," in *Black Hat DC*, 2009.

[37] C. Evans and C. Palmer, "Public Key Pinning Extension for HTTP," IETF, Internet-Draft draft-ietf-websec-key-pinning-03, Oct. 2012.

[38] A. Langley, "Public key pinning," http://www.imperialviolet.org/2011/05/04/pinning.html.

[39] C. Paya, "Certificate pinning in Internet Explorer with EMET," http://randomoracle.wordpress.com/2013/04/25/certificate-pinning-in-internet-explorer-with-emet/.

[40] M. Marlinspike and E. T. Perrin, "Trust Assertions for Certificate Keys," IETF, Internet-Draft draft-perrin-tls-tack-02, Jan. 2013.

[41] I. Dacosta, M. Ahamad, and P. Traynor, "Trust no one else: Detecting MITM attacks against SSL/TLS without third-parties," in *Proceedings of the European Symposium on Research in Computer Security*, 2012.

[42] "Certificate Patrol - a psyced Firefox/Mozilla add-on," http://patrol.psyced.org.

[43] M. Dietz, A. Czeskis, D. Balfanz, and D. Wallach, "Origin-bound certificates: A fresh approach to strong client authentication for the web," in *Proceedings of the 21st USENIX Security Symposium*, 2012.

[44] D. Wendlandt, D. Andersen, and A. Perrig, "Perspectives: Improving SSH-style host authentication with multi-path probing," in *Proceedings of the USENIX Annual Technical Conference*, 2008.

[45] M. Marlinspike, "SSL and the future of authenticity," in *Black Hat USA*, 2011.

[46] K. Engert, "DetecTor," http://detector.io/DetecTor.html.

[47] M. Alicherry and A. D. Keromytis, "Doublecheck: Multi-path verification against man-in-the-middle attacks," in *IEEE Symposium on Computers and Communications*, 2009.

[48] R. Holz, T. Riedmaier, N. Kammenhuber, and G. Carle, "X. 509 forensics: Detecting and localising the SSL/TLS men-in-the-middle," in *Proceedings of the European Symposium on Research in Computer Security*, 2012.

[49] P. Eckersley, "Sovereign key cryptography for internet domains," https://git.eff.org/?p=sovereign-keys.git;a=blob;f=sovereign-key-design.txt;hb=master.
[50] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," IETF, Internet-Draft draft-laurie-pki-sunlight-02, Oct. 2012.
[51] T. H.-J. Kim, L.-S. Huang, A. Perrig, C. Jackson, and V. Gligor, "Accountable Key Infrastructure (AKI): A proposal for a public-key validation infrastructure," in *Proceedings of the International Conference on World Wide Web*, 2013.
[52] R. Sleevi, "[cabfpub] Upcoming changes to Google Chrome's certificate handling," https://cabforum.org/pipermail/public/2013-September/002233.html, 2013.
[53] P. Hoffman and J. Schlyter, "The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA," RFC 6698 (Proposed Standard), Internet Engineering Task Force, Aug. 2012.
[54] P. Hallam-Baker and R. Stradling, "DNS Certification Authority Authorization (CAA) Resource Record," RFC 6844 (Proposed Standard), Internet Engineering Task Force, Jan. 2013.