View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by JScholarship

Classifying Network Protocol Implementation Versions: An OpenSSL Case Study

Paul D. Martin
Johns Hopkins University

Michael Rushanan
Johns Hopkins University

Stephen Checkoway
Johns Hopkins University

Matthew Green
Johns Hopkins University

Aviel D. Rubin
Johns Hopkins University

Abstract

A new technique is presented for identifying the implementation version number of software that is used for Internet communications. While many programs may exchange version numbers, oftentimes only a small subset of them send any information at all. Furthermore, they usually do not provide accurate details about which implementation is used. We use machine learning techniques to build a feature database and then apply this to network traffic to try to identify specific implementations on servers. We apply our technique to OpenSSL and report our results.

1 Introduction

Many Internet protocols communicate meta information about the software that is running at each end host. Included are details such as protocol version number, available parameters, software name and version, and other useful information that allows protocols to self-tune their interaction. Although the meta information included in the early messages in Internet protocols can be extremely useful, it also introduces potential security risks. Due to publicly available lists of software vulnerabilities corresponding to early versions of nearly any well-known implementation, by announcing an implementation version number, the software is also broadcasting a set of vulnerabilities. Although exchanging protocol version number is frequently necessary to ensure interoperability, as a security precaution, many protocol implementations provide no information about the *implementation* version number. This represents a challenge to those interested in measuring and classifying Internet traffic.

Our goal in this paper is to provide a framework and a tool for measuring the prevalence of protocol implementation versions and specific instantiations for common communication protocols on the Internet, without trusting the meta information shared by the communicating parties. Rather than believing the version numbers and implementation identifiers that are included in the traffic, we attempt to automatically infer protocol version numbers and to additionally classify network traffic according to specific implementations, based on observable implementation fingerprints.

Our approach is to use machine learning to extract features from training data for known protocol implementations and versions. Once we build a database of features for a particular protocol, X, we crawl the web, speaking X to as many protocol peers as possible. Our analysis engine then uses the database of features that was built with the training data to attempt to determine what version and implementations of X we have found. Thus, we are able to measure the prevalence of specific protocol versions and implementations on the Internet without trusting that the implementations are speaking the truth.

As a case study, we implement our system and measure the prevalence of OpenSSL versions on the Internet. We find that a small fraction (about 7%) of Apache deployments use a Linux distribution-default configuration which report the OpenSSL version used. We use this as ground truth data to train our classifiers. Our results indicate that many insecure versions of OpenSSL are in active deployment, and we believe that our research presented here has led to a much more accurate analysis of the state of SSL on the Internet than was previously possible. This case study demonstrates the utility of our general framework for measurement.

2 Tool design and case study methodology

In this section we give a high-level overview of our classification tool, which uses a combination of benign scanning and machine learning to identify version numbers for protocol implementations deployed on remote servers. We designed the tool using a modular architecture so that it may be used in a variety of Internet measurement studies. The end result is a tool which is useful for classifying protocol implementations for which we have some labeled

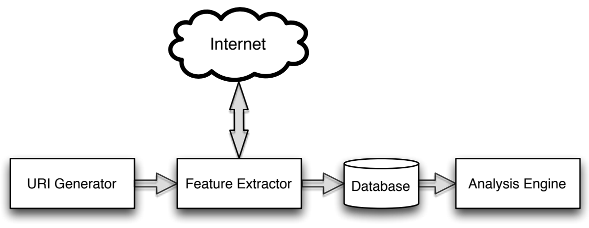

Figure 1: Architectural diagram for the classification tool.

data. For our SSL/TLS case study, we rely on web servers which are configured to report their version number.

Our classification tool consists of four components: a URI generator, a feature extractor, a database, and an analysis engine. Figure 1 shows the components and the flow of information through the system. The URI generator, feature extractor and classification algorithm are modular and can be easily replaced by new modules for different tasks. We first describe each component in the abstract; then we describe our concrete instantiation for classifying OpenSSL versions.

URI generator. The URI generator's job is to produce a set of URIs to be consumed by the feature extractor. This modular component can be as simple as a static set of URIs produced one at a time or as complex and dynamic as may be required for the particular use case. The same generator can be used for a variety of classification tasks. Examples of generators include a generator that produces the Alexa top 500 sites [1], a generator that walks through the entire IPv4 address space, and a web crawler that starts at a fixed set of web sites and recursively crawls links it encounters. We use the last example in our SSL/TLS case study.

Feature extractor. The feature extractor takes as input a URI, interacts with the network server pointed to by the URI, and produces a feature vector for an arbitrary set of features. For example, the feature extractor can fetch web pages, query the server for configuration options, or engage in aggressive probing such as that performed by standard network measurement tools like nmap. The feature extractor is necessarily specific to the classification task; however, the modular nature of our architecture makes swapping out the feature extractor for a different implementation easy.

Database. After the feature extractor has produced a feature vector, it is inserted into a generic relational database.

Analysis engine. The heart of our system is the analysis engine which runs one of a variety of machine learning classification algorithms on the data in the database. The analysis engine heavily leverages the Python Orange library [22], an open source data visualization and analysis tool, to first cluster and then classify the data. The analysis engine assumes that some subset of the data has labels.

We used a semi-supervised learning approach to classify the remaining, unlabeled data. Since there are likely to be a large number of protocol implementation versions (e.g., for our OpenSSL study, we found 79 distinct versions), we use *k*-nearest neighbors (*k*NN) as our classifier. The downside of *k*NN is that its accuracy depends heavily on the similarity metric used. To mitigate this, we first perform a *k*-means clustering and then run *k*NN for each cluster.

Although we use *k*-means and *k*NN, they are not the only possible choices. Some applications may perform better with other choices and thus this is a configurable option. We experimentally validated that *k*-means and *k*NN are good choices for our use case among the set of algorithms supported by Orange. Another option which may perform well for our use would be a multiclass logistic regression since many of our features are binary or categorical.

2.1 OpenSSL version instantiation

To be useful as a classification tool, the URI generator and feature extractor need concrete instantiations.

URI generator. Our URI generator is implemented as a web crawler that traverses HTTP links and recursively looks for any new HTTPS links to pass to the feature extractor. To determine where to start our crawl, we first identified several websites that contain a large number of external links to many different domains. We pointed our crawler at several of these sites, including `http://www.cnn.com` and `http://news.ycombinator.com` and ran it until we had collected a set of about 123,000 websites. We believe this set to be a representative sample of the sites frequently visited by average users because the sites are all within a few degrees of separation from popular websites.

Feature extractor. For each URI produced by the URI generator, the feature extractor initiates a secure connection to the server and interrogates it to learn a variety of features including:

- supported SSL/TLS versions and ciphersuites;
- secure/insecure renegotiation support;
- session resumption support;
- CA, validity, issue date, expiration date;
- TLS compression support;
- TCP/IP stack information;
- HTTP headers;
- web server configuration.

Some of the features we discovered had surprising characteristics. For example, we discovered that the server response HTTP header was reported by 81.7% of the servers in our data set. 7.8% of those server responses contained a number of useful features such as the version numbers of most of the Apache software stack including Apache modules like mod_perl and mod_php and, importantly for us, OpenSSL. These version numbers form the labels that the analysis engine uses. The server response is described in more detail in Section 4.

We construct our feature extractor by extending version 0.4 of iSECPartner's SSLyze tool [15]. SSLyze provides methods to enumerate all cipher suites for SSLv2, SSLv3, TLSv1.0, TLSv1.1 and TLSv1.2. In addition, SSLyze provides methods to enumerate session resumption support, to scan for insecure session renegotiation, and to collect information about site certificates (e.g., expiration date, issuing certificate authority, and certificate signature validation).

We implemented the following extensions to SSLyze via new plugins: TLS compression support detection (i.e., to determine vulnerability to the CRIME attack [14]); HTTP headers collection generated by website navigation; server response string parsing and TCP/IP configuration information collection.

3 SSL/TLS background

Secure Socket Layer (SSL/TLS) [7] is perhaps the most important security protocol on the Internet. Although SSL/TLS has many applications — including Virtual Private Networking and inter-application communication — it is most commonly used to secure web traffic served via the HTTPS protocol. The relevance of TLS and HTTPS has increased in recent years, as many websites (including Gmail, Facebook and Twitter) have begun to deploy HTTPS by default [6, 9, 11, 25].

While TLS is widely deployed across the Internet, there are only a small number of popular implementations. When considering HTTPS, the most common are OpenSSL (used by a majority of Apache deployments) and Microsoft's SChannel. Results from the February 2013 Netcraft survey indicate that Apache servers dominate IIS implementations across surveyed Internet domains [18], indicating that OpenSSL is likely the most popular TLS implementation on the Internet.

In fact, given the widespread dependence on OpenSSL, it is reasonable to say that for many users, OpenSSL *is* TLS. This should draw attention from the security community as there are numerous versions of OpenSSL in current deployment, many of which include serious protocol vulnerabilities [3, 19, 29]. If an attacker can determine the version of OpenSSL deployed on a specific webserver, he may be able to seriously compromise the effectiveness of any TLS connection made to that site.

Surprisingly, we have very little information on the deployment of OpenSSL library versions, because most web servers do not advertise this information. Even if an Apache server version is known, this does not necessarily imply that the server is using a known version of OpenSSL: in many systems, the installed OpenSSL version is determined by various factors, including the particular Apache and OS distribution, the presence of other software on the system, and (most commonly) server misconfiguration. This can lead to the widespread deployment of old, broken versions of the library.

3.1 SSL/TLS (in)security

Recent attacks on SSL/TLS, such as BEAST [10] and CRIME [14], demonstrate the sensitivity of SSL/TLS security to configuration choices made by server administrators. We briefly describe several protocol and implementation vulnerabilities below. Many of these were identified by the SSL Pulse project [24].

SSLv2 support. SSLv2 is known to have many vulnerabilities which make it unsuitable for use in secure communications. Although SSLv2 is considered so insecure that major Linux distributions don't even build OpenSSL with support for it anymore, the SSL Pulse data estimates that 28.4% of SSL servers on the Internet today support this version of the protocol, more than the number of sites that support TLSv1.1 and TLSv1.2 combined (9.2% and 11.4% respectively) [24].

Insecure session renegotiation. In 2009 a practical attack on SSLv3/TLSv1.0 was proposed that exploited flaws in the session renegotiation feature allowing for a man-in-the-middle attack. The vulnerability allows an attacker to queue an HTTP command to be executed by the server on behalf of the client immediately when a client makes an SSL connection.

Insecure CBC mode ciphersuites. Several recent attacks have illustrated flaws in the implementation of CBC mode encryption within SSL and TLS. These flaws stem from two basic causes: the improper use of initialization vectors [4, 10] and flaws related to TLS's insecure MAC-then-encrypt approach to authenticated encryption [3, 5]. The latest of these vulnerabilities [3] was patched in February, 2013, hence these attacks remain an active concern. Similar flaws are also present in Datagram TLS, and were recently exploited by AlFardan and Paterson [2].

TLS compression support. A general class of attacks exploiting protocol vulnerabilities due to TLS compression and SPDY. While this mode of operation has been recognized as insecure for a number of years [16], the recent CRIME attack demonstrated the feasibility of using this vector to attack critical information such as session cookies [14].

Software vulnerabilities. The OpenSSL implementation of TLS and SSL has been subject to numerous software vulnerabilities. These include dozens of implementation flaws with potential consequences ranging from denial of service to remote code execution [21].

One of the results of our research is the ability to identify versions of OpenSSL in use on the Internet. Since many versions of OpenSSL contain known, severe vulnerabilities, such as information leakage and remote code execution [26–28], by examining a large, representative sample of servers, we get an indication of what fraction of servers in use today are vulnerable.

4 Results

All tests described below were run on m2.4xlarge Amazon EC2 instances with 64 GB of RAM and four cores running Red Hat Enterprise Linux 6, PostgreSQL 8.4.12, Python 2.6.6 and Orange 2.0b. The tests were extremely memory and CPU intensive and even with these sizable resources our tests took on the order of hours to days to run on a test data set containing 123,345 scanned sites.

4.1 Prediction accuracy

As described in Section 2, our analysis engine takes a semi-supervised learning approach to version identification. To determine how accurate our classification is, we perform a 10-fold cross validation on our training data which consists of the self-reported OpenSSL versions. Interestingly, we find that if we exclude *only* the OpenSSL version from the Apache server response HTML header, we can identify the OpenSSL version with overwhelming probability by using the reported version numbers of Apache modules like mod_php and mod_perl; however, this result is misleading. Among all 9,653 of the scanned web sites which report module versions numbers, only 615 do not report an OpenSSL version. We hypothesize that this is due to the use of a different SSL/TLS library such as GNU TLS. Thus, when testing accuracy, we omit the entire server string from each 10% test set.

Because the accuracy of kNN is so dependent on the similarity metric, we first cluster similar points and run kNN on each cluster (see Section 2). The accuracy of our classification (weakly) depends on the number of clusters we use. We test with $k = 10$, 20, and 30 clusters. For each cluster, we use $k = 10$ neighbors to classify each test point. OpenSSL versions are split into: major, minor, fix, patch and (optionally) distro-build versions [20]. For example, an OpenSSL version running on Fedora Linux might be openssl-1.0.0a-fedora1. The major version would be 1, the minor 0, the fix 0, the patch a and the distro-build fedora1. For OpenSSL, there are essentially two major.minor version combinations in use, 0.9 and 1.0. Therefore, we tread major and minor versions as a single version component. Table 1 and Figure 2 show the percentage of versions that can be identified correctly.

Version Component	$k = 10$	$k = 20$	$k = 30$
major.minor	91.6	94.3	93.4
fix	75.5	74.0	75.7
patch	50.2	49.6	54.7
distro-build	49.6	49.2	54.7

Table 1: Percentage of OpenSSL version components correctly predicted for k clusters. The percentages in each row are the percentages of correctly predicting the version components up to the given component.

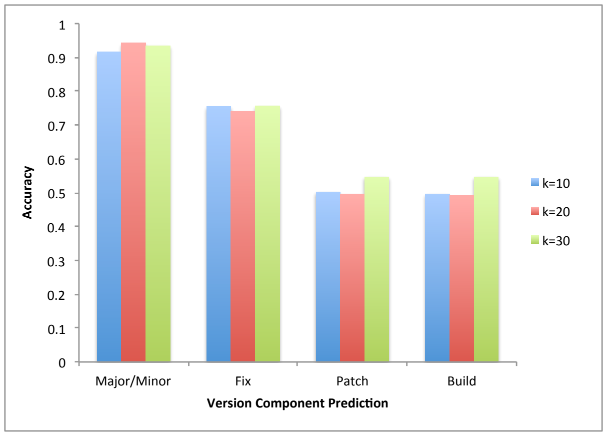

Figure 2: Percentage of OpenSSL version components correctly predicted for k clusters.

4.2 Observed OpenSSL versions

Since we are focused on classifying OpenSSL versions, we need to remove data points which do not correspond to OpenSSL versions but come from Microsoft SChannel, GNU TLS, or other SSL/TLS implementations. The first step of the analysis engine is to perform k-means clustering. This produces clusters of data whose feature vectors lie close together in the feature space. We hypothesize that OpenSSL data points are likely to cluster together (although not necessarily in a single cluster) and that other implementations are not likely to lie in clusters with (many) OpenSSL data points. To that end, for each cluster, we examine the number of data points that are labeled and throw out any cluster which does not contain at least 10% labeled data.

After removing clusters with too few labeled data points, we have 61,832 unlabeled data points. From this we can give a rough breakdown of popularity of OpenSSL versions on the Internet, Table 2. In the following section, we discuss the implications of these results in the context of vulnerabilities present in different versions of OpenSSL.

Version	Percentage
0.9.8e-fips-rhel5	37.25
0.9.8g	14.50
0.9.7a	7.02
0.9.8o	4.76
1.0.0-fips	4.36
0.9.7d	2.91
0.9.8n	2.75
0.9.7e	1.94
0.9.8c	1.80
0.9.8m	1.74
0.9.8e	1.72
0.9.8r	1.71

Table 2: Most popular OpenSSL versions on the Internet.

Distribution	OSSL Version	CVEs
Debian Squeeze (6.0)	0.9.8o	11
Debian Lenny (5.0)	0.9.8g	24
Debian Etch (4.0)	0.9.8c	26
RHEL 6	0.9.8e/1.0.0-fips	0/14
RHEL 5	0.9.7a/0.9.8e-fips	14/0
RHEL 4	0.9.6b/0.9.7a	9/14
Fedora 18	1.0.1c	3
Fedora 17	1.0.0i	3
Fedora 16	1.0.0e	9

Table 3: Default OpenSSL versions shipping with popular Linux distributions.

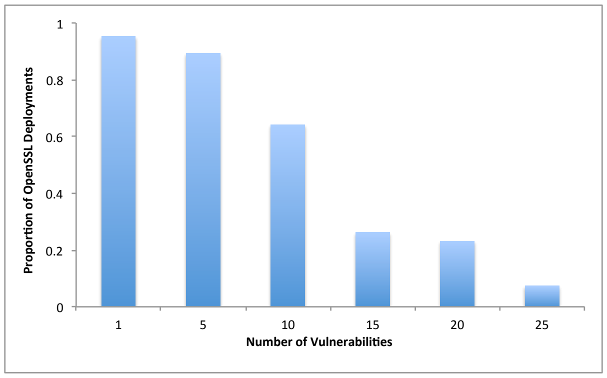

Figure 3: Percentage of OpenSSL deployments with at least *n* unpatched vulnerabilities

4.3 Vulnerabilities

From the OpenSSL website's known vulnerability report section [21] we see that 75.9% of known vulnerabilities apply to five or more OpenSSL versions within the same patch family. Thus if the OpenSSL implementation can be accurately fingerprinted to within a patch family then there is a high probability that at least some of the vulnerabilities reported for the guessed OpenSSL version will apply even if the guess is incorrect.

Based on the results in Table 2 and data from the OpenSSL vulnerability report [21], we have computed the number of reported vulnerabilities for each OpenSSL version that we have predicted using our analysis engine. Our results show that 95% of all deployed OpenSSL versions have at least one known CVE that hasn't been patched by OpenSSL (but that may have been patched by an individual distribution vendor). 64.12% of all deployed OpenSSL versions have more than 10 CVEs. The complementary CDF of vulnerabilities is presented in Figure 3.

One of the least surprising findings of our survey is that most users are running the OpenSSL version included in their Linux distribution and, in many cases, users do not keep their Linux installations up to date. We will discuss potential reasons for this in the next section. Table 3 includes a list of three of the most popular Linux distributions that we found in our crawl as well as the default OpenSSL version that each shipped with.

We note that many of the most popular OpenSSL versions in Table 2 are versions that shipped in the popular Linux distributions in Table 3. This table indicates that most of the default OpenSSL versions included in shipping Linux distributions have several known vulnerabilities. It is important to note that Linux distributions may patch some of these vulnerabilities on their own, though manufacturers usually do not patch all vulnerabilities. This is because many vulnerabilities are discovered after support for a given version is no longer being supported. Our training data is unable to take these patch levels into account for many distributions (such as Debian). This is a fundamental limitation of our data set and of the training data available. As a result, the number of CVEs filed against an OpenSSL version form an upper bound of how many known vulnerabilities might exist, but in practice the number of outstanding vulnerabilities could be lower. However, simply discovering the base version of a particular OpenSSL server raises the possibility of it being vulnerable to a known attack.

5 Discussion

The results of our SSL/TLS case study validate our modular framework and general approach for identifying and classifying versions of network protocol implementations. Because our framework treats the modular components (the URI generator, feature extractor, and classification algorithms) as black boxes, we can plug in different modules to study other protocol implementations or a different swath of the Internet with a minimum of effort. In the future, we plan to do exactly this (see Section 8).

In the rest of this section, we discuss the significance of these results and address some of the questions that they raise.

5.1 Severity of vulnerabilities

Our results indicate that many of the OpenSSL servers on the Internet are vulnerable to implementation-specific exploits described in CVEs cataloged by the OpenSSL project. Our first consideration is the severity of the vulnerabilities. Of the 54 vulnerabilities, four can lead to remote code execution, fourteen to a DoS and seven to information leakage. We observe that types of vulnerabilities range in severity from moderate to catastrophic. We therefore assert that not only are these OpenSSL servers vulnerable to many of the protocol attacks as described by the SSL Pulse project [24] but that they are also widely unprotected against implementation-specific attacks that could, in some cases, not only compromise the privacy of an existing SSL session, but also the security of the underlying server.

5.2 Distribution-specific patches

We have also observed that many Linux distributions release package updates in fixed-length cycles. The versions of all major libraries, including OpenSSL, are fixed at the end of distribution's release cycle. The this prevents all further package updates, including security patches, that the library vendor releases from being automatically integrated into new builds of the package. If a major vulnerability is discovered an individual distribution's package maintainer may elect to backport the patch and release it as part of a security update for that distribution, but this is not guaranteed.

This process is problematic for several reasons. First, the package maintainers responsible for backporting patches cannot necessarily provide the level of scrutiny toward patch as the library maintainers can. Also, many of the CVEs are released after a distribution has stopped releasing security patches for an outdated, but still widely deployed release. This results in mission-critical systems being left connected to the Internet without any ability to receive security patches. To illustrate this point, our analysis has revealed that Debian Lenny which officially supports OpenSSL version 0.9.8g is still widely deployed on the Internet. This particular version has 24 known vulnerabilities, 17 of which Debian patched before it discontinued support for Lenny. The remaining vulnerabilities will never be patched. We found that 0.7% of all of the servers that we crawled self-reported that they run Lenny in the server response header. An attacker viewing this information will immediately know that this server must be running unpatched versions of many libraries and daemons that may be vulnerable to remote code execution exploits.

5.3 TLS1.1/1.2 deployment

The only OpenSSL branch that supports TLS1.1/1.2 is the OpenSSL 1.0.1 branch [20], which, as shown in Table 2 is not widely deployed. We believe that administrators are hesitant to upgrade to the 1.0.1 branch of OpenSSL because most Linux distributions have not yet released packages for it. To further complicate matters, the OpenSSL, mod_ssl, and Apache versions are interdependent making upgrading one without the others difficult. These complications prevent many administrators from upgrading their web server stacks on their own. Since most Linux distributions do not include the OpenSSL 1.0.1 branch, a large fraction of Apache servers cannot support the most recent versions of TLS, even if their administrators wanted to. This observation is corroborated by the SSL Print's observations of TLS1.2/1.2 market penetration during the same time period as our crawl [24]

5.4 Case study conclusions and future work

The current state of OpenSSL deployment on the public Internet is not good. By actively interrogating web servers, our tool discovered close to 62% of web servers using OpenSSL are running versions which contain known vulnerabilities that are susceptible to exploitation. This is explained, at least in part, by the fact that many web server stacks use the default version of OpenSSL provided by the operating system which can become stale if not kept up-to-date.

In the future, we plan to explore the nature of biases in our training data by leveraging additional classification techniques on additional features such as operating system version. We believe that the additional data labels will provide insight to our current semi-supervised approach. In addition, we plan to explore avenues for acquiring better training data for additional TLS implementations.

6 Limitations of data

The approach that we use for collecting our training data is a good approach for fingerprinting OpenSSL implementations but it has several limitations. The major limitation of our approach comes from our analysis making the assumption that the observed labels (the OpenSSL version numbers) come from web servers that accurately report them. We believe that this is likely to be the case since Apache programatically generates this information when it is run rather than being hard coded in a configuration file. A hard coded value would run the risk of not being updated when OpenSSL is updated and thus the server would report old version numbers. Another concern is that a clever administrator might try to spoof these labels to throw off would-be attackers. While this is a concern, we did not observe any cases where observed features did not match the reported OpenSSL version. Additionally, there is the possibility that our training data may be skewed towards servers that are configured with default settings or towards servers that are infrequently updated. Both concerns can be addressed by starting with a better set of data for which we have ground truth.

We are also limited by the fact that we are only able to collect training data for OpenSSL and thus cannot identify other SSL/TLS implementations such as GNU TLS or Microsoft SChannel. As a result, we were forced to rely on clustering techniques to identify likely OpenSSL implementations from among the servers we crawled. This introduces both false positives — that is, SSL/TLS libraries we classify as being OpenSSL when they are not — and false negatives — throwing out OpenSSL data points because the clusters do not contain enough labeled instances. Since we again do not have ground truth, we cannot measure the accuracy of this binning technique. More complete training data would remove this limitation.

Finally, we were only able to scan web servers on the public Internet. Thus, our results do not generalize to all OpenSSL deployments such as those on corporate LANs or behind NAT gateways, an important problem in its own right due to the presence of malware on these networks resulting from, among other things, bring-your-own-device corporate cultures.

7 Related work examining SSL/TLS

Much of the SSL security research to date considers the quality of certificates deployed across websites, as well as server-based configuration options. Community driven projects such as the EFF SSL Observatory and Netcraft SSL Survey have been erected to provide an in-depth analysis of server certificates on the web. More concretely, the EFF SSL Observatory aims to determine the trustworthiness of Certificate Authorities by investigating multiple features of all server certificates on the web (e.g., number of certificates signed by an authority) [8]. The Netcraft SSL Survey is a monthly data collection service that attempts to identify how online businesses use encryption to secure their online transactions (i.e., confirming known certificate usage and deployment) [18]. In 2012, Heninger et al. [12] performed an Internet-scale analysis of all SSL certificates. The intent of the analysis was to determine how many servers had weak or insecure key generation mechanisms. The end result was the startling realization that 5.57% of deployed SSL servers shared an RSA key factor.

In the past few years there have been several high profile attempts to survey SSL servers that are vulnerable to the known protocol attacks with varying areas of focus [13, 15, 17, 23]. In contrast, we take an entirely new approach to analyzing the security of SSL servers by looking at individual, implementation-specific vulnerabilities rather than looking for protocol-specific ones. In doing so we classify the OpenSSL implementations of a large sample of servers and chart these implementations against known version-specific vulnerabilities for which CVEs exist. We show that a large percentage of servers are running OpenSSL versions for which there are many unpatched vulnerabilities. In many cases, these vulnerabilities are severe and in some cases lead to total privacy breaches or remote code execution [26–28].

Finally, recent work attempts to fingerprint SSL *library type* (e.g., SChannel vs. OpenSSL) by actively probing TLS handshake responses [30] for a list of known error responses. While this tool, SSLAudit, can distinguish different libraries, it is not able to determine library version.

8 Conclusions and future work

We have shown that machine learning techniques can be effective as a means of classifying Internet communication based on the implementations that generated the traffic. Identifying specific implementations is extremely useful for measuring and analyzing security. For example, in our case study looking at SSL/TLS, we discovered that more than 62% of the installations running OpenSSL deployed versions that are known to be susceptible to published exploits.

Our case study discovered that default configurations are typically maintained, and we know that these often become stale quickly. Studies such as ours can be used to identify the prevalence of patch applications on the Internet, and we believe that our technique can be easily applied to other protocols. We plan to continue to enhance our machine learning techniques and to implement additional case studies to study the implementations of other security-sensitive protocols such as SSH or DNS. One objective of this work is to provide measurement tools that analysts can use to discover how widely patches have been applied to different software packages. We also hope our work will provide generic tools for researchers who need to learn about the use of particular implementations of protocols.

References

[1] Alexa The Web Information Company. Alexa top sites. `http://www.alexa.com/topsites`, 2013.

[2] Nadhem J. AlFardan and Kenneth G Paterson. Plaintext-recovery attacks against datagram TLS. In *Proceedings of NDSS 2012*, 2012.

[3] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *Proceedings of IEEE Symposium on Security and Privacy ("Oakland") 2013*. IEEE Computer Society, 2013.

[4] Gregory V. Bard. A challenging but feasible blockwise-adaptive chosen-plaintext attack on SSL. In *Proceedings of SECRYPT 2006*, pages 7–10. INSTICC Press, 2006.

[5] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In *Proceedings of CRYPTO 2003*, pages 583–599. Springer, 2003.

[6] Lucian Constantin. Most of the Internet's top 200,000 HTTPS websites are insecure, trustworthy internet movement says. *CIO Magazine*, 2012.

[7] Tim Dierks and Eric Rescorla. The transport layer security (TLS) protocol version 1.2. http://www.ietf.org/rfc/rfc5246.txt, 2008.

[8] Electronic Frontier Foundation. The EFF SSL Observatory. https://www.eff.org/observatory, 2010.

[9] Facebook. The Facebook blog: A continued commitment to security. https://www.facebook.com/blog/blog.php?post=486790652130, 2011.

[10] Dan Goodin. Hackers break SSL encryption used by millions of sites. http://www.theregister.co.uk/2011/09/19/beast_exploits_paypal_ssl/, 2011.

[11] Google. Official Gmail blog: Default https access for Gmail. http://gmailblog.blogspot.com/2010/01/default-https-access-for-gmail.html, 2010.

[12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your Ps and Qs: Detection of widespread weak keys in network devices. In *Proceedings of USENIX Security 2012*. USENIX, 2012.

[13] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. The SSL landscape: A thorough analysis of the x.509 PKI using active and passive measurements. In *Proceedings of IMC 2011*. ACM Press, 2011. http://doi.acm.org/10.1145/2068816.2068856.

[14] iSEC Partners. Details on the "Crime" attack. https://www.isecpartners.com/news-events/news/2012/september/details-on-the-crime-attack.aspx, 2012.

[15] iSECpartners. iSEC partners releases SSLyze. https://www.isecpartners.com/tools/application-security/isec-partners-releases-sslyze.aspx, 2013.

[16] John Kelsey. Compression and information leakage of plaintext. In *Proceedings of FSE 2002*, pages 263–276. Springer, 2002. http://www.iacr.org/cryptodb/archive/2002/FSE/3091/3091.pdf.

[17] McAfee. SSLDigger McAfee v1.02. http://www.mcafee.com/us/downloads/free-tools/ssldigger.aspx, 2013.

[18] Netcraft. Netcraft: February 2013 web server survey. http://news.netcraft.com/ssl-survey/, 2013.

[19] OpenSSL Project. OpenSSL security advisory, 2007. http://www.openssl.org/news/secadv_20071012.txt.

[20] OpenSSL Project. OpenSSL: Cryptography and SSL/TLS Toolkit. http://www.openssl.org/, 2013.

[21] OpenSSL Project. Openssl vulnerabilities. http://www.openssl.org/news/vulnerabilities.html, 2013.

[22] Orange. Orange website. http://orange.biolab.si/, 2013.

[23] Travis Saltman. Fingerprinting SSL tutorial. http://travisaltman.com/fingerprinting-ssl-tutorial/, 2013.

[24] Trustworth Internet Movement. SSL Pulse: Survey of the SSL implementation of the most popular web sites. https://www.trustworthyinternet.org/ssl-pulse/, 2013.

[25] @twitter. Securing your Twitter experience with HTTPS. http://blog.twitter.com/2012/02/securing-your-twitter-experience-with.html, 2012.

[26] US-CERT/NIST. OpenSSL CVE-2007-5135. http://www.cvedetails.com/cve/CVE-2007-5135/, 2007.

[27] US-CERT/NIST. OpenSSL CVE-2007-4995. http://www.cvedetails.com/cve/CVE-2007-4995/, 2007.

[28] US-CERT/NIST. OpenSSL CVE-2010-3864. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3864, 2010.

[29] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 debian openssl vulnerability. In *Proceedings of IMC 2009*. ACM Press, 2009. http://doi.acm.org/10.1145/1644893.1644896.

[30] Thierry Zoller. SSL Audit - TLS/SSL Scanner. http://www.g-sec.lu/sslaudit/documentation.pdf, 2013.