Securing SSL Certificate Verification through Dynamic Linking

Adam Bates
CISE Dept.
University of Florida
Gainesville, FL
adammbates@ufl.edu

Joe Pletcher
CIS Dept.
University of Oregon
Eugene, OR
pletcher@cs.uoregon.edu

Tyler Nichols
CIS Dept.
University of Oregon
Eugene, OR
tnichols@cs.uoregon.edu

Braden Hollembaek
iSEC Partners
Seattle, WA
bhollemb@cs.uoregon.edu

Dave Tian
CISE Dept.
University of Florida
Gainesville, FL
tian@cise.ufl.edu

Kevin R. B. Butler
CISE Dept.
University of Florida
Gainesville, FL
butler@ufl.edu

ABSTRACT

Recent discoveries of widespread vulnerabilities in the SSL-/TLS protocol stack, particular with regard to the verification of server certificates, has left the security of the Internet's communications in doubt. Newly proposed SSL trust enhancements address many of these vulnerabilities, but are slow to be deployed and do not solve the problem of securing existing software. In this work, we provide new mechanisms that offer immediate solutions to addressing vulnerabilities in legacy code. We introduce CertShim, a lightweight retrofit to SSL implementations that protects against SSL vulnerabilities, including those surveyed by Georgiev et. al. [19], in a manner that is transparent to the application. We demonstrate CertShim's extensibility by adapting it to work with Convergence, DANE, and Client-Based Key Pinning. CertShim imposes just 20 ms overhead for an SSL verification call, and hooks the SSL dependencies of 94% of Ubuntu's most popular packages with no changes necessary to existing applications. This work significantly increases system-wide security of SSL communications in non-browser software, while simultaneously reducing the barriers to evaluating and adopting the myriad alternative proposals to the certificate authority system.

Categories and Subject Descriptors

C.2.0. [**Computer-Communication Networks**]: General — *Security and protection*

Keywords

SSL, TLS, HTTPS, public-key certificates

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

http://dx.doi.org/10.1145/2660267.2660338.

1. INTRODUCTION

Internet applications have relied for years on the SSL/TLS libraries for secure end-to-end channels, but a growing body of literature points to systemic lapses in security procedure that renders our communications insecure. The heart of these problems lies with the inability for clients to accurately authenticate the server when presented with its public-key certificate. Certificate validation has been shown to be incorrect at all layers of the SSL stack, from improper certificate handling in libraries [7, 29], to confusion and abuse of SSL APIs [19, 43], to applications that are *broken by design* so that they are easier to use [17]. Moreover, high profile compromises of prominent Certificate Authorities (CAs) have eroded the very foundations of the SSL trust model [12, 18, 32]. Any one of these lapses gives rise to the threat of a Man-in-the-Middle (MitM) attack, in which an attacker is able to intercept and read supposedly-secure SSL traffic in transit to or from a target website.

While a variety of forward-thinking solutions have been proposed in the literature [10], less attention has been paid to immediate countermeasures, and ways in which we can fix the vast amount of legacy software that is vulnerable and inexorably linked to the CA model. Detecting and reporting these vulnerabilities is an inadequate solution; studies of Android SSL usage have found that up to 76% of vulnerabilities persist for over a year [43], even once the developers have been notified of confirmed vulnerabilities [17]. In the presence of aloof and unavailable developers, we must pursue alternate means of securing our Internet communications.

In this paper, we consider a system-wide approach to securing negligent SSL code in non-browser software that simultaneously facilitates the immediate use of CA alternatives and other SSL trust enhancements. We present CertShim, a lightweight retrofit to existing popular SSL implementations (OpenSSL, GnuTLS, and partial support for JSSE) that acts as a *shim*[1] into the validation procedure in dynamically linked applications. CertShim provides application and domain specific handlers that *force correct cer-*

[1]A shim is a library that transparently intercepts an API and changes its parameters or operations.

tificate validation on previously insecure applications. We demonstrate its practicality by incorporating a variety of verification techniques including traditional CA validation, Convergence [30], DANE [23], and client-side key pinning [14, 31, 37, 42], some of which were previously only available as experimental browser plug-ins. CertShim reduces the barrier to adopting these systems by making them immediate candidates for system-wide deployment.

While recent studies have made recommendations for the general improvement of the SSL ecosystem, few have introduced system-wide defenses to SSL vulnerabilities in legacy software. Fahl et al. modify the Android API to restrict SSL misuse [17], while Conti et al. introduce MYTHIS, a benign MitM proxy for Android that is able to defend against rogue access point attacks [11]. In contrast to these works, our platform-wide defense does not require a manufacturer update, or even administrator privileges, to be put to use. We also deliver on the promise of *pluggable certificate verification* that is left to future work by Fahl et al., and go a step further by showing that additional security assurances can be attained by layering multiple verification modules. CertShim works in desktop and server environments, which are considerably more complex than Android, as various SSL implementations need be considered. Furthermore, the solutions for Android experience compatibility issues with some programs; in contrast, we present a policy engine that provides application- and domain-specific certificate handling.

We make the following contributions:

- **Enforce Safe Defaults on SSL:** CertShim hooks calls to SSL APIs in order to enforce hostname validation and certificate validation. As this behavior would otherwise break applications that pin certificates or connect to domains that use self-signed certificates, we present a policy engine that enforces safe defaults but provides unique handlers based on the application and destination domain.
- **Enable CA Alternatives:** Existing open source initiatives may be slow to adopt alternative SSL trust models, may intentionally choose to stay with the status quo of certificate authorities, or may simply have gone defunct. CertShim provides modular retrofits that allow existing applications to use modern CA alternatives. CertShim provides a means of taking the consensus of multiple forms of certificate validation, which to our knowledge is the first of its kind in the literature, allowing even stronger guarantees through ensemble validation.
- **Performance Analysis of CertShim:** We survey Ubuntu's 10,000 most popular packages and find that CertShim supports 94% of the 390 packages that were found to contain SSL usage. Our benchmarks show that the use of CertShim adds as little as 20 ms to an SSL lookup under realistic conditions. We also perform manual testing to determine that CertShim transparently secures all of the major SSL library misconfigurations and 8 of the 9 data-transport library vulnerabilities identified by Georgiev et al.[19].

The remainder of this work is organized as follows: Section 2 provides background on SSL, CAs, and the problems associated with certificate validation. In Section 3 we present the design and implementation of CertShim, analyze its features in Section 4, and evaluate its performance as well as coverage of real world SSL usage in Section 5. Limitations of our approach and future work are discussed in Section 6, and related work is summarized in Section 7. In Section 8 we conclude.

2. BACKGROUND

The SSL/TLS protocol families are largely responsible for securing the Internet's web traffic. The original SSL (Secure Socket Layers) protocols were introduced by Netscape in 1995 to provide confidentiality, integrity, and identity to network communications [22]. While the foundations of SSL's solutions to confidentiality and integrity have withstood the test of time, reliably establishing destination identity in SSL connections has proven to be a surprisingly difficult problem. Without identity assurances, users are vulnerable to the threat of impersonation or Man-in-the-Middle (MitM) attacks, in which an attacker is able to intercept and read supposedly-secure SSL traffic bound to or from a target website. We still rely on Netscape's original solution, the Certificate Authority (CA) public key infrastructure, which requires that domains register with one or more CAs in exchange for a signed X.509 certificate. A client can then authenticate the server validating its certificate by using the issuing CA's public key; obtaining this public key is a transparent process to the client, as it is likely pre-installed into their operating system or web browser.

Development Vulnerabilities. Good SSL code, particularly with regards to certificate verification, is very difficult to correctly implement. Numerous MitM vulnerabilities have been discovered in certificate chain validation routines, such as null prefix attacks on Pascal strings [28, 29]. Georgiev et al's survey of SSL connection authentication exposed pervasive misunderstanding of certificate verification in SSL API's in non-browser software, and also drew attention to many popular SSL libraries that are broken by design [19]. Fahl et al. interviewed developers to find that apps were often intentionally broken so as to allow for easier development, or to support self-signed certificates [17]. Even worse, the tomes of vulnerable SSL code that exist today that are unlikely to ever be patched. Studies of SSL vulnerabilities in Android apps have found that up to 76% of vulnerabilities persist for over a year [43], even once the developers have been notified of confirmed vulnerabilities [17]. Even more surprisingly, Brubaker et al. perform automated testing to uncover dramatic inconsistencies in the validation routines of the major SSL libraries, some of which gave rise to exploitable vulnerabilities [7]. Given that not even SSL library development teams can agree on best practices for certificate handling, the situation today is certainly grim.

Trust Vulnerabilities. Orthogonal to these implementation issues are a number of fundamental and systemic limitations in the CA trust model. CAs are under no obligation to perform due diligence before issuing a certificate, and in fact this lack of verification is pervasive in the certificate ecosystem [45]. This, combined with the myriad recent serious, high-profile compromises and blunders (e.g., Comodo[32], Diginotar[18], TURKTRUST[12]), makes it fair to ask whether CAs are sufficiently incentivized to preserve the security of themselves or their customers. These lapses are at times met without any serious repercussions [9, 30], and there is even evidence that CAs work directly against

customer security by offering wiretap services [21, 39, 41, 42]. These problems are serious enough, but they are exacerbated by a lack of scoping; any CA can verify any certificate, meaning that conscientious businesses that certify with reputable CAs are just as at risk, and software vendors often include potentially untrustworthy CAs in order to ensure compatibility with Internet services [15].

2.1 CA Alternatives

Proposed CA enhancements and alternatives were surveyed by Clark and van Oorschot [10], who identify *families* of proposals based on their underlying fundamental principles of operation. These alternatives vary widely in terms of both their advantages and limitations, reflecting differences of opinion on the fundamental problems with the CA trust model. This work makes use of an important subset of these security enhancements that require no server side changes in order to be adopted; primitives such as multipath probing [3, 30, 47], client-based key pinning [14, 31, 37, 42], and certificate revocation lists [13, 26, 33, 38] are eligible for immediate deployment by individual users, providing tangible security enhancements to today's Internet threats. Due to its relative popularity, we also consider the DANE DNS-based key pinning system, a trust enhancement that embeds X.509 certificates in DNSSEC records [23, 27]. However, none of these trust enhancements has enjoyed widespread deployment, in part due to the vast amount of non-browser software that would need to be modified in order to enable their system-wide use.

Key pinning, Convergence, and DANE are exemplar trust enhancements that we make use of in this work. However, each suffers from inherent design or trust limitations that impact their applicability in certain scenarios. For example, client-based key pinning cannot determine whether a change in a server's certificate is malicious or benign. Recent work by Amann et al. [2, 4] shows that routine changes to SSL trust relationships are often indistinguishable from attacks, making this a noteworthy limitation. In contrast, Convergence can offer insight into the cause of the unexpected certificate by detecting whether the certificate has changed globally, or just locally. However, if the new certificate is the result of a MitM attack near the server, the Convergence notaries will conclude that the change is benign, resulting in a dangerous false negative. DANE, by checking the target domain's TLSA record, could offer a definitive answer to whether the change was benign, but DANE is being incrementally deployed and further bloats the traditional CA trust model to include the DNS architecture. In this work, we show that by querying multiple certificate validation systems it is possible to retain the benefits of each while mitigating each system's limitations.

3. DESIGN

3.1 Threat Model & Assumptions

Our system is designed with consideration for an adversary in the network that attempts to launch a MitM attack against SSL communications; this attack could be launched from a rogue wireless access point to which the client is connecting, or from elsewhere on the Internet, such as somewhere on the network paths between the client and server. CertShim works under multiple trust models, and we therefore consider adversaries of varying strengths. A weak attacker might only possess an untrusted CA certificate, but a stronger attacker might possess a valid certificate from a trusted CA, or even control parts of the Internet architecture (e.g., network paths, trusted CAs, DNS resolvers). We discuss the security of various verification modules against these adversaries in Section 3.4. When client-based key pinning is considered, we assume that the Client has had one opportunity to correctly authenticate the server in the past, which is necessary for use in *Trust on First Use (TOFU)* authentication [5].

We seek to secure client-side non-browser applications that are benign but potentially defective in their use of SSL. In particularly, we wish to protect against *insecure use* of SSL libraries. Except where otherwise noted, we do not consider incorrectness within the underlying libraries themselves [7, 20], assuming instead that they are correctly implemented. This work is motivated by the fact that developers often fail to fix vulnerable code, so we assume that the applications will not take active countermeasures to bypass our mechanism. CertShim interposes on popular known SSL libraries; we do not consider applications that use their own SSL implementations. Doing so would increase the cost and complexity of software development, and in evaluation we found no evidence that this was a widespread practice. However, we do anticipate that applications can use many layers of abstraction/misdirection in calling an SSL API, such as a cURL wrapper for a scripting language.

3.2 System Goals

We set out to design a mechanism that provided the following system-wide properties:

Override Insecure SSL Usage. Force safe defaults for certificate handling (i.e., validation of hostnames and certificate chains) on all SSL connections, regardless of whether or not the application makes any attempt to do so. This encompasses applications that misconfigure the SSL API, applications that use insecure SSL wrappers, and applications that are broken by design.

Enable SSL Trust Enhancements. In addition to traditional verification, our system should be configurable to enable the use of CA alternatives and enhancements. CA Alternatives are often incapable of correctly authenticating *all* Internet domains due to design limitations or incremental deployment, so we also wish to provide a means of querying multiple verification methods and reconciling their results.

Maximize Compatibility. Due to the great diversity of SSL usage, existing software will use SSL APIs in ways that we cannot anticipate, some of which may be perfectly valid and secure. These constraints could be application-specific, such as using a pinned certificate, or trusting a corporate CA. Others could be domain-specific, such as a server on an unreachable private network, or a server that has not published a TLSA record [23]. Our mechanism must be able to coexist with these applications without breaking them.

Maximize Coverage. As applications could conceivably re-implement SSL from the RFCs, it is impossible to enumerate all of the possible SSL libraries. However, we wish to maximize the coverage of our defensive layer by supporting the libraries that are most commonly used by SSL applications in practice.

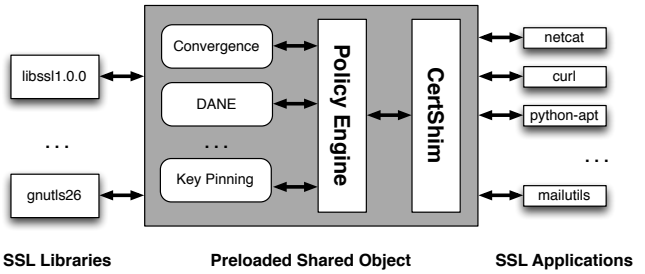


Figure 1: CertShim interoperates with various SSL libraries, replacing their certificate verification functions with those from proposed CA alternatives.

Function	Location	Purpose
connect	libssl1.0.0	Initial SSL handshake
do_handshake	libssl1.0.0	Renegotiate handshake
get_verify_result	libssl1.0.0	Check verification result
handshake	libgnutls26	SSL handshake
certificate_verify_peers2	libgnutls26	Verify certificate (deprec.)
certificate_verify_peers3	libgnutls26	Verify certificate
CheckIdentity	JDK6	Verify hostname
CheckIdentity	JDK7	Verify hostname
SetEndpointIdentifAlg	JDK7	Verify hostname
connect	System call	Track hostname, port
gethostbyname	System call	Track hostname, port
getaddrinfo	System call	Track hostname, port

Table 1: Functions that CertShim overrides/hooks. Multiple hooks are required because libraries have multiple entry points that trigger certificate verification.

```
1  int SSL_get_verify_result(SSL *ctx)
2    // Determine SSL initialization type
3    type = resolve_ctx_type(ctx)
4
5    // Obtain the domain and port
6    name = lookup_name(ctx)
7    port = lookup_port(ctx)
8
9    // Grab the certificate fingerprint
10   sha = extract_ctx_fingerprint(ctx)
11
12   // Call a validation Function
13   status = 0
14   if (CONFIG_CERT_PINNING)
15     status += keypin_verify(name,port,sha)
16   if (CONFIG_CERT_AUTHORITY)
17     status += ca_verify(name,port,sha)
18   if (CONFIG_CONVERGENCE)
19     status +=
           convergence_verify(name,port,sha)
20   if (CONFIG_DANE)
21     status += dane_verify(name,port,sha)
22
23   // Check the results
24   if (resolve(status) == OK)
25     return X509_V_OK
26   else
27     return X509_ERR_INVALID_CA
```

shim.c

Figure 2: Pseudocode for CertShim's dynamically loaded `SSL_get_verify_result` function.

3.3 CertShim

Our system, CertShim, is a dynamically linked shared object that performs binary instrumentation in order to layer additional security onto popular SSL implementations, OpenSSL and GnuTLS, and includes an additional mechanism that instruments JSSE's `SSLSocketFactory`. Shown in Figure 1, CertShim works primarily through use of the Linux dynamic linker's `LD_PRELOAD` environment variable. Under normal circumstances, a function such as `ssl_connect()` would resolve to code in `libssl.so`. However, when CertShim is enabled, the linker first looks at our shared object before moving to the standard include paths, allowing us to redefine the behavior of the library's verification function without modifying specific executables, as shown in Figure 3. CertShim performs additional certificate checks and also calls the original functions such that the security semantics of the original library workflow are preserved. If those additional checks fail, CertShim triggers a connection failure and also creates a `syslog` record that includes both an explanation and a suggested template policy for fixing the problem. The specific form of verification is modular, and is discussed at greater length in Section 3.3.2.

3.3.1 Function Hooks

CertShim targets both *verification functions* and *handshake functions* in SSL libraries as well as system calls that allow for the recovery of important *network context*. A list of the CertShim's function hooks is included in Table 1.

Verification Functions. CertShim instruments OpenSSL's `ssl_get_verify_result()` and GnuTLS' `gnutls_certificate_verify_peer()` to support modular certificate verification. Regardless of which module is enabled, the return values of the instrumented function are consistent with the original library API. Pseudocode for CertShim's version of `ssl_get_verify_result()` can be found in Figure 2. In lines 2-7, the canonical name and port of the certificate are recovered. In lines 9-10, the certificate fingerprint (SHA1 hash) is extracted from the SSL context. Certificate validation functions are called in lines 12-19. Lines 21-25 return the standard X509 accept or reject values based on whether or not the certificate was approved.

Handshake Functions. Hooking the verification functions alone is insufficient to force proper verification in vulnerable applications due to the fact that they often go unused by developers [19]. Therefore, CertShim also targets the main connection functions of SSL libraries, which represent a choke point at which we can force certificate verification. We instrument OpenSSL's `ssl_connect()` and `ssl_do_handshake()` functions and GnuTLS's `gnutls_do_handshake()` function. CertShim first calls the original functions from each respective library, returning its error code if the connection failed on its own (e.g., due to network connectivity failure). If the call is successful, however, CertShim calls the verification module as described above. If verification fails, CertShim generates an error that emulates a connection failure, essentially short-circuiting the SSL connection and forcing the application to recover (shown in Figure 3). This system behavior is likely to cause unexpected behavior in some applications; however, as we show in Section 3.4 the behavior of the verification module can be configured to prevent application breakage.

CertShim's inclusion of both handshake and verification functions is to ensure that vulnerable SSL code is hooked. This is necessary because numerous studies have shown that

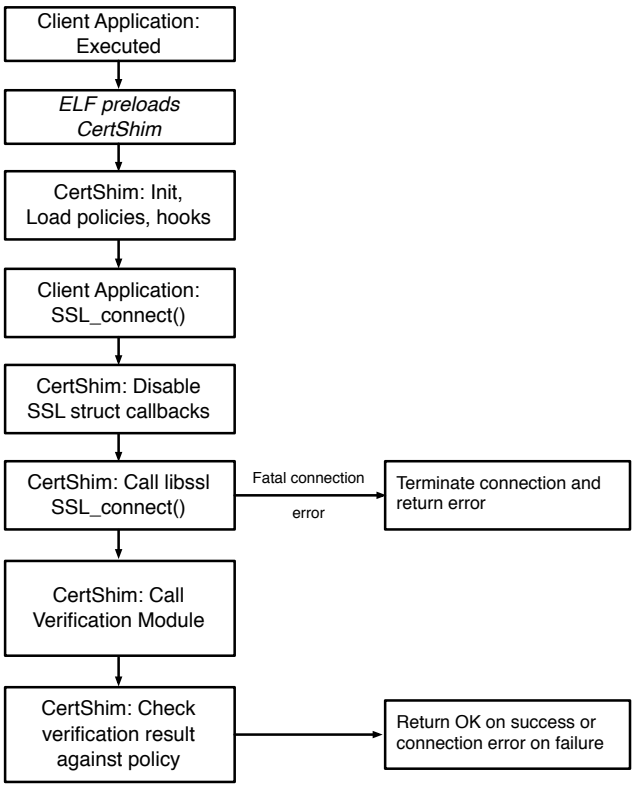


Figure 3: Interaction between a client application using the `SSL_connect()` function and CertShim.

negligent developers often fail to consult SSL verification functions [17, 19]. As a consequence, however, CertShim will redundantly verify certificates in well-formed SSL code. The verification function hooks are superseded by the handshake hooks in normal usage, but we felt it important to hook the verification functions so that the libraries' certificate handling was consistent across different parts of the API. We show in Section 5 that the performance costs of using CertShim are minimal, and that this redundancy is a small price to pay for the added coverage assurances.

Network Context. Some alternate certificate verification methods, such as network probing, require a canonical hostname and port in order to validate an X509 certificate. However, the structures passed into OpenSSL and GnuTLS functions do not reliably contain this information. This is due to the great variety of ways in which these routines are invoked; in some cases, certificates are verified without being aware of the endpoint with which the SSL session is being established. To recover this information, we instrument the `getaddrinfo()`, `gethostbyname()`, and `connect()` system calls. By recording the parameters passed and returned from the original functions, we were able to perform reverse lookups that translated socket file descriptors to hostnames, which were then stored in a `sqlite3` database that was keyed from the process id of the calling function. Keying off the pid was important to allowing the database to be shared between processes, preventing file descriptor collisions.

3.3.2 Verification Modules

Currently, CertShim supports 4 certificate verification methods:

- **Traditional CA Verification**: The module invokes the underlying native SSL API calls.
- **Convergence**[30]: The module communicates with Convergence Notaries via a REST API. Convergences local cache is implemented as a `sqlite3` database, and the list of trusted notaries and a verification threshold is set via a configuration file.
- **DANE** [23]: The module is a thin wrapper around Lexis' SWEDE library for TLSA record verification [35]. This serves to demonstrate that existing prototypes for SSL trust enhancements can be easily adapted for use with CertShim. With minor modifications, this module could be used to deploy Liang et. al.'s DANE extension for securing CDN connections in non-browser software [27].
- **Client-Based Key Pinning**: Implementations currently exist as Firefox Plug-Ins [37, 42]. Rather than adapting these utilities, we developed our own trust-on-first-use key pinning module that stores certificate fingerprints in an `sqlite3` database.

Not only does CertShim facilitate the use of any one of these modules, it also offers support for certificate validation through ensemble voting strategies. For example, all 4 of the modules can be enabled simultaneously, with a majority vote determining whether or not the certificate is approved. In the event that a verification method fails but overall verification passes, CertShim prints a notification of the failure to `syslog`. Because CertShim uses a single verification module across many implementations of SSL, prototyping clients for new CA alternatives becomes a one-time cost and interoperability with a variety of SSL libraries is assured.

3.4 Policy Engine

CertShim includes a policy engine that allows users to easily express complex certificate verification routines that can be enforced system-wide or tailored to a specific application or domain. Policies are defined by the user in an Apache-like configuration file that is dynamically loaded every time an application is invoked, allowing for the user to alter the SSL behavior of all applications on the system at any time without having to recompile. Figure 4 is an example of such a policy definition file. The configuration subsystem of CertShim uses `libconfig` [1] for parsing the configuration file and extracting data.

The structure of the policy file is easy to understand and use. A `global_policy` entry defines the system-wide behavior of CertShim except for cases where a more specific policy is present. The engine finds relevant policy entries by pattern matching over the `cmd` and `host` keys, with priority being given to host entries. The `methods` key in each entry allows for the enabling and disabling of specific verification modules. The `vote` key represents the percent of modules that must return true before CertShim approves the certificate. When a key is not set in a specific entry, it inherits the value of the global policy. The ordering of the policies within the configuration file is irrelevant.

```
global_policy: {
  cert_pinning   = false;
  cert_authority = true;
  convergence    = false;
  dane           = false;
  vote           = 1.00;
};

command_policies: ({
  cmd = "/usr/bin/git";
  vote = 1.00;
  methods: {
    cert_authority = false;
    convergence    = true;
  };
}, {
  cmd = "/usr/bin/lynx";
  vote = 0.50;
  methods: {
    cert_pinning   = true;
    convergence    = true;
    dane           = true;
  };
});

host_policies: ({
  host = "www.torproject.org";
  vote = 1.00;
  methods: {
    cert_authority = false;
    dane = true;
  };
});
```

config.cfg

Figure 4: A sample CertShim policy configuration file

Figure 4 shows an example configuration file that illustrates the granularity and flexibility of the CertShim policy configuration engine. Here, the global policy is set to force traditional CA verification on all SSL connections. However, this user connects to a GitLab versioning server that makes use of a self-signed certificate, so they created a command policy entry for `git` that uses the Convergence module. The user also wants stronger assurances than traditional CAs can provide when browsing with Lynx, so they create an additional command entry that queries all 4 modules and requires that at least 2 return true. This entry inherits its `cert_authority` value from the global policy. Finally, when connecting to domains that are known to offer DANE support such as `torproject.org`, the user adds a host policy entry that requires DANE validation. Policies that tolerate failure in this fashion are useful when using CertShim as a platform for testing CA alternatives that may not be entirely stable.

We envision that various stakeholders will share the responsibility of CertShim policy creation, alleviating the burden on end users. While some users may wish to define their own certificate management policies, software development communities could also release application-specific policies, similar to the manner in which `selinux` policy modules are included in software packages. Operating system development communities that make use of CertShim could also include a policy of safe defaults in their distribution.

3.5 Java Instrumentation

In the Java architecture, we cannot interpose on SSL libraries such as JSSE and BouncyCastle through Linux's dynamic linking. Instead, we make use of the `java.lang.instrumentation` interface to achieve similar functionality inside of the JVM. We successfully used this method to provide CertShim-like functionality by hooking the `checkIdentity()` function in JDK 6's JSSE, and the `checkIdentity()` and `setEndpointIdentificationAlgorihtm` function of JDK 7. Georgiev et al. point to misuse of the low level JSSE `SSLSocketFactory` API, which does not perform hostname verification, as one of the biggest SSL vulnerabilities in Java [19]. While in control of these functions, CertShim overrides the applications' configuration in order to force hostname verification. Like our C-based mechanism, our instrumentation object can be injected into all Java calls by setting an environment variable. Java is not yet fully supported in CertShim, as we have not re-implemented our policy engine. However, this mechanism demonstrates how our approach can be generalized to work with Java.

4. ANALYSIS

We now consider the extent to which CertShim meets our 3 primary system goals: *override insecure SSL usage*, *enable SSL trust enhancements*, and *maximize compatibility*. We consider the extent to which we achieve *maximal coverage* in Section 5.

4.1 Override Insecure SSL Usage

Recent work has uncovered strong evidence that insecure certificate handling practices are often a result of developer confusion and apathy [16, 17, 19]. Rather than wait on developers, CertShim automatically fixes these vulnerabilities without requiring developer intervention. Enforcing safe defaults for SSL does not even require policy configuration, as CertShim installs with a global default policy that enforces CA verification. We also include fail-safe protections to the policy engine, such as the `vote` key defaulting to 1.00 if left accidentally unspecified by the user.

CertShim supports all applications that dynamically link to OpenSSL and GnuTLS, two of the most popular open source SSL libraries. In Section 5, we show that this provides support for 94% of SSL usage in the most popular Ubuntu packages. Most excitingly, CertShim fixes certificate verification in data-transport libraries that are *broken by design*, including `urllib`/`urllib2`, `httplib`, python's `ssl` module, and perl's `fsockopen` call. This aspect of CertShim proves critical, as the survey in Section 5 finds that such libraries represent up to 33% of SSL usage in Ubuntu packages.

4.2 Enable SSL Trust Enhancements

In this work, we implement verification modules for 3 exemplar CA alternatives, making them immediate candidates for system-wide deployment. Switching from CA verification to an alternative such as Convergence requires a change of just 2 lines in the CertShim configuration file. Due to incremental deployment or design limitations, some CA alternatives are not universally applicable to the entire SSL ecosystem. For example, not all HTTPS domains have published TLSA certificates for DANE, and other domains will be inside closed networks that cannot be verified with Convergence's multi-path probing. We have further contributed

to the adoptability of CA alternatives by introducing a policy engine that allows for application and domain specific certificate handling. With CertShim, it is possible to force DANE verification only on domains that are known to be supported. CertShim even helps to support traditional CA verification by providing a multi-path probe module that can be enabled specifically for applications and domains that make use of self-signed certificates.

4.2.1 Consensus Verification

CertShim further improves SSL security by providing the first practical means of reconciling the results of multiple certificate verification handlers. In so doing, it is possible to overcome practical problems or trust concerns that are limitations of different architectures. To demonstrate the power of this approach, we present sample policy entries that represent unique trust and usage models for SSL. For a detailed explanation of the security properties of these systems, please refer to the original works. We believe that the combining of different verification primitives through consensus voting represents a promising new direction for securing SSL.

Distrust the CAs. Convergence was motivated by the goal of completely removing certificate authorities from the SSL trust model. CAs are replaced with notaries, trusted third parties that are incentivized to be trustworthy agents due to *trust agility*, the ability of the user to change who their trusted notaries at any time. However, multi-path probing cannot validate all domains. One option would be to combine Convergence with client-based key pinning:

```
cert_pinning = true;
convergence = true;
vote = 0.50;
```

This configuration allows for a CA-free trust model. When Convergence is unable to validate a domain, CertShim would default to a trust-on-first-use model [5]. In the event that a certificate is updated, in most cases Convergence would be able to re-validate the domain, making up for key pinning's inability to offer context in the event of a benign anomaly. In the event of a discrepancy between a cached certificate and the certificate presented by the host, the key pinning module would fail to verify the certificate, requiring the user to manually decide whether to trust the presented certificate.

Server-side MitM Defense. Convergence relies on network path diversity in order to validate certificates. While this is adequate for detecting local MitM attacks at rogue access points, if a powerful adversary such as a nation state can control all paths between the server and the notaries, Convergence could yield a false negative during an attack. To account for this possibility, CertShim could tether its trust to the DNS architecture:

```
convergence = true;
dane = true;
vote = 1.00;
```

This policy increases attack complexity by requiring the attacker to control the DNSSEC resolvers and a valid certificate from a trusted CA in addition to all network paths to the server. We note that this policy only works for domains that offer DANE support. In environments where DNSSEC is actively being used, the use of Convergence provides a hedge against DNSSEC server compromises.

4.3 Maximize Compatibility

As the invasiveness of our function hooks increased, so too did the likelihood that CertShim would break applications. Developers could not have anticipated our layering of additional certificate verification methods on top of their code. Applications may disable certificate verification in order to support self-signed certificates [17], contact domains that are not compatible with certain forms of verification, or even have implemented their own security features such as key pinning. As a result, CertShim's actions could trigger unexpected behavior.

These realities motivated the creation of our policy engine, which offers the ability to completely eliminate compatibility issues by performing application and domain specific certificate handling. Regardless of the policy in effect for a given connection, the CertShim hook return values strictly adhere to the OpenSSL and GnuTLS APIs. This implies that existing applications are unable to detect the presence of CertShim while allowing CertShim to remain entirely method-agnostic. That is, CertShim does not interfere with the logic built into existing applications since return values remain true to the OpenSSL and GnuTLS APIs and CertShim itself holds no opinion on which verification methods it should or should not use; CertShim can even be configured to take no action for a given application or domain. Furthermore, the success or failure of alternate verification methods is translated into return codes consistent with OpenSSL and GnuTLS.

5. EVALUATION

In this section, we evaluate CertShim for both its ability to support real world SSL usage and the performance costs it imposes on SSL connections.

5.1 Coverage

Our investigation of CertShim coverage is comprised of two parts. We first perform a small-scale survey in which we manually test applications and libraries to confirm support, followed by a large-scale survey in which we conduct semi-automated source code inspection to estimate CertShim coverage for a fuller distribution of software.

5.1.1 Manual Testing

Our evaluation of CertShim coverage began with manual testing of popular SSL applications and middleware. Presently, CertShim is confirmed to support 12 different SSL implementations or wrappers, shown in Table 2. Although it was apparent that the listed SSL scripting wrappers all used OpenSSL/GnuTLS backends, it was necessary to manually confirm compatibility because the wrappers occasionally made use of the SSL API in unexpected ways. For example, we discovered that CertShim does not support `php_curl` due to the fact that this library statically links `libcurl`. Continuing with manual testing, we selected a handful of common SSL applications to confirm CertShim support, shown in Table 3. Of these, CertShim successfully hooked each application except for Firefox, which is due to the fact that Mozilla uses LibNSS rather than OpenSSL or GnuTLS. We discuss the broader implications of these coverage gaps in Section 6.

Program	Success	Confirmed With
libcurl	Yes	C program
gnutls26	Yes	C program
libssl1.0.0	Yes	C program
SSLSocketFactory	Yes	java program
perl socket::ssl	Yes	perl script
php_curl	No	php script
fsockopen	Yes	php script
httplib	Yes	python script
pycurl	Yes	python script
pyOpenSSL	Yes	python script
python ssl	Yes	python script
urllib, urllib2	Yes	python script
gnutls-cli	Yes	CLI execution

Table 2: Libraries and wrappers that were manually confirmed to be supported by CertShim.

5.1.2 Less Dangerous Code?

We next consider CertShim's safe default features, and how they protect against the SSL vulnerabilities presented by Georgiev et al. [19]. Through manual testing, we confirmed that CertShim would secure the SSL communications in 100% of the SSL libraries, 89% of the data-transport libraries, and 71% applications mentioned in this work:

- **SSL Libraries**. Error-prone aspects of the SSL API are identified in OpenSSL, GnuTLS, and JSSE. CertShim enforces proper certificate handling for OpenSSL and GnuTLS, even when the application fails to call the verification function. We provide partial support to JSSE, which we instrumented to ensure that hostname verification is always performed, regardless of how the API is invoked.
- **Data-transport Libraries**. Georgiev et al. discuss 9 data-transport frameworks that wrap the major SSL libraries. We provide full support to `cURL`, php's `fsockopen`, `urllib`, `urllib2`, `httplib`, `python ssl`, and partial support to the Java libraries Apache HttpClient and Weberknect. The only library that CertShim does not support is `php_curl` due to static linking.
- **SSL Applications**. We obtained the vulnerable versions of several of the applications explored, including Lynx, Apache HttpClient, and Apache Axis. Based on review of these applications combined with our manual tests, we conclude that CertShim secures the SSL communications of 12 of the 17 applications mentioned that could be run on Linux systems. All 5 of the unsupported applications were payment services that had a vulnerable `php_curl` dependency.

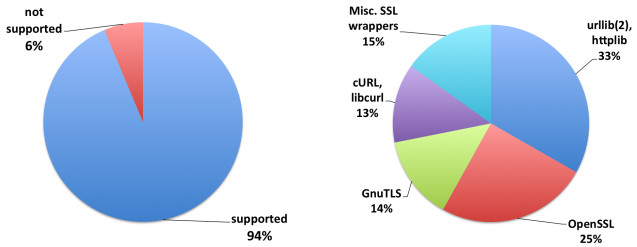
These findings demonstrate the power of the CertShim methodology. It also serves to show that, so long as the API remains the same, CertShim can protect against presently undiscovered vulnerabilities and misconfigurations in SSL APIs and wrapper libraries.

5.1.3 Large-Scale Coverage Survey

On a general computing platform such as Linux, automated dynamic analysis of SSL proved difficult due to the great variety of SSL implementations, languages, and usage scenarios. During manual testing, we found that a thorough knowledge of an application's purpose and behaviors was required in order to trigger SSL connections. As a result, we were unable to perform large-scale dynamic analysis of CertShim. Fortunately, based on the supported libraries shown in Table 2, we were able to use a mix of static analysis and manual inspection in order to arrive at a CertShim coverage estimate for a large corpus of applications.

Program	Test Cmd	Success
curl	curl https://google.com	Yes
sslscan	sslscan google.com:443	Yes
lynx	lynx -dump https://google.com	Yes
ncat	ncat –ssl-verify google.com 443	Yes
fdm	*Checked gmail.com over SSL*	Yes
fetchmail	*Checked gmail.com over SSL*	Yes
firefox	*Visited gmail.com (w/o plugin)*	No
mpop	*Checked gmail.com over SSL*	Yes
perl	*Perl's IO::Socket::SSL*	Yes
pycurl	*cUrl established SSL session*	Yes
pyOpenSSL	*Socket established SSL session*	Yes
urllib	*urllib made HTTPS request*	Yes
w3m	w3m https://google.com -dump	Yes
wget	wget https://google.com	Yes
gnutls-cli	Performed handshake procedure	Yes

Table 3: Programs and libraries that were manually tested to confirm CertShim support



(a) Total Packages Supported (b) Packages supported by SSL library/wrapper

Figure 5: Estimated CertShim support of SSL usage found in Ubuntu Popularity Contest.

As a source for real world SSL usage, we selected the Ubuntu Popularity Contest [34], a service that tracks the most commonly installed packages on Ubuntu. Starting with the top 10,000 packages, we ran `apt-rdepends`, a tool for recursively finding library dependencies for a given package, on each package. Of these packages, we were able to recover library dependency information for 7,789 packages due to the fact that some packages were not present in the main apt repositories for Ubuntu 12.04. From this list we discovered 2,949 packages that had dependencies to known SSL libraries (i.e., the packages left out *could not* have been using SSL). We then gathered the corresponding source files using `apt-get source`. This methodology yielded 1,809 codebases, with the reduction in total packages being accounted for by the fact that one codebase can be responsible for multiple packages. With the available source, we proceeded to check the files against a list of keywords related to networking, SSL, and HTTPS. After narrowing the field to source packages containing keywords, we manually inspected the remaining packages to confirm that the package made SSL connections. As CertShim only partially supports Java, these packages that were removed from the survey.

Dependency	Support?	Pkg. Count
`urllib/httplib`	Yes	123
`OpenSSL`	Yes	92
`GnuTLS`	Yes	51
`cURL/libcurl`	Yes	48
Misc SSL Wrappers	Yes	56
Total Supported		370
Total Unsupported		26

Table 4: Details of estimated CertShim support and SSL usage in the Ubuntu Popularity Contest.

Module	Real Time
OpenSSL w/o CertShim	88 ms [84, 92]
CertShim Baseline	108 ms [107, 109]
Convergence Baseline	108 ms [107, 110]
Key Pinning, First Use	130 ms [120, 139]
Key Pinning, Revisit	119 ms [118, 119]
DANE	7 sec

Table 5: Benchmarks for CertShim usage. 95% confidence intervals are included in brackets.

Because CertShim cannot support static linking, we also wished to determine if this was a common practice. To do so, we installed the 395 packages that contained SSL activity, then ran `ldd`, a tool that prints shared library dependencies, against each of the resulting files that were placed in `bin` or `lib` directories. `ldd` lists packages that are dynamically linked, and can also detect static linking. We did not find widespread use of static linking, of 10,707 files checked by `ldd`, we found only 12 that were statically linked.

As illustrated in Figure 5a, we found that CertShim supported 370 of the 395 packages found to be making SSL connections, for a coverage ratio of 94%. Our use of the word "support" can be interpreted as follows – this application *may* make an SSL connection in execution, and if it does CertShim*will* hook it. A stronger assurance about application behavior would have required dynamic analysis, which was not feasible. A summary of our results can be found in Table 4 and Figure 5b. The miscellaneous SSL wrappers included QSSL, Pidgin's Purple SSL, URLGrabber, Serf, and Neon; for each, we inspect the source code to confirm that they wrapped OpenSSL or GnuTLS and used one of CertShim's function hooks. The unsupported packages included previously discovered coverage gaps such as NSS, as well as other wrappers such as KSSL and QCA, for which we were unable to confirm support. We note that this coverage result is an estimate; without dynamic analysis, it was impossible to definitively confirm that these applications attempted to make SSL connections. However, we did confirm that each of the applications had code paths that made web requests with SSL-ready libraries.

5.2 Performance

We generated several benchmarks for the baseline performance of CertShim, performing tests on a Dell PC running a Linux 3.5 kernel with 2 GB of RAM and a Pentium 3Ghz dual-core processor. We measured the time it took the `wget` utility to retrieve a small, 9 KB file over HTTPS from a nearby web server. This call triggers the `SSL_get_verify_result()` function, which is supported by CertShim. The throughput of the connection to the server was approximately 80 MB per second. The server was using a CA-signed certificate, which was validated by `wget` during the course of the download. Each of these results were averaged over 500 measurements. When CertShim was not loaded, `wget` returned in 88 *ms*. When CertShim was loaded without a verification module, the operation completed in 108 ms, imposing just 20 ms base overhead on OpenSSL. This overhead is largely due to the CertShim hooks for the `connect()` and `getaddrinfo()` functions, which collect contextual data that is required by the hooked OpenSSL and GnuTLS functions and write it to a SQLite database. The policy engine demonstrated an average run time of just 0.061 ms while parsing the sample configuration file and initializing the policies. When using an 86 kilobyte configuration file consisting of 392 policies, the policy engine required an average run time of 3.075 ms. Each average was based on 1,000 iterations.

We then repeated these trials with the different verification modules enabled. The results are summarized in Table 5. The minimum time required for Convergence verification was 108 ms, corresponding to the case in which the client already possesses a locally cached copy of the certificate fingerprint. We benchmarked Key Pinning under two use cases: verification took 130 ms when visiting a domain for the first time, and 119 ms when checking a previously visited domain. The time required to use the DANE module was 7 seconds. We attribute this exorbitant cost to the fact that our DANE measurements used `https://www.torproject.org` instead of a local server. Tor has 9 IP addresses associated with this domain, each of which was sequentially verified by the SWEDE library within our module.

Initially, we observed that the base cost of CertShim was 900 ms. Upon further investigation, we realized that this was due to our SQLite configuration; each attempt to open a write transaction to the database cost approximately 100 ms. To improve performance, we disabled journaling on the database. We note that this also disabled protections against database corruption due to hardware failures or unexpected interrupts. In a future iteration of CertShim, we intend to restore these protections while imposing minimal additional performance cost by implementing an in-memory database that flushes to disk during idle periods.

6. LIMITATIONS & FUTURE WORK

We now discuss several potential gaps in CertShim's coverage, as well as possible solutions:

Root Processes. For security reasons, `LD_PRELOAD` is not permitted by default for processes running as root; however, support for root can be provided by symbolically linking a root-owned copy of CertShim to the `/lib` directory.

Alternative Libraries. CertShim supports modern versions of two of the most popular open source SSL libraries, `libssl1.0.0` and `gnutls26`. There are many other implementations available, such as `PolarSSL` and `NSS`; we have inspected these libraries and believe that CertShim can be extended to support them with only modest additional work. CertShim provides a blueprint for interposing on function calls in any C-based SSL library. Our approach could be deployed on Windows' `SChannel` library as an `AppInit_DLL`.

SSL in Java. Linux dynamic linking cannot be used to interpose on Java libraries. However, in Section 3.5 we demonstrate that our approach is applicable in Java through instrumentation objects.

Static Linking. As our methodology is based on dynamic linking, CertShim cannot interpose on statically linked executables. In our evaluation, we encountered one instance of static linking in PHP's cURL wrapper. As a stop-gap solution to this problem, we are investigating the use of Conti et al.'s approach of correlating SSL traffic to applications [11]; if a flow's network context is not already present in the CertShim database (Section 3.3.1), CertShim can infer that the flow is from a statically linked application and warn the user. This limitation could be more fully addressed via static binary instrumentation through use of tools like PEBIL [25] and DynInst [6, 8].

Usability. CertShim already has a complete logging system, making it easy to develop a graphical user interface layer and other utilities. We envision a notification system similar to Red Hat's `setroubleshootd` daemon, which simplifies `selinux` usage by alerting users to new AVC log messages. Because CertShim logs a template policy entry upon verification failure, an `audit2allow`-like application could be developed to dramatically simplify policy debugging.

Policy Engine. We intend to extend the policy mechanism to support pattern matching on command line arguments. Some applications' certificate verification behaviors should change based on these arguments; for example, CertShim should take no action when `wget` is invoked with the `-no-check-certificate`. We also intend to release additional verification modules.

7. RELATED WORK

A large body of recent work has sought to better understand and ultimately prevent SSL vulnerabilities in non-browser software. Georgiev et al. manually survey different layers of the SSL stack, discovering pervasive misconfigurations of certificate validation routines, as well as usage of SSL libraries that are broken by design [19]. Brubaker et al. specifically target SSL libraries, generating 8 million random permutations of valid X509 certificates to perform differential testing and discover hundreds of certificate validation discrepancies [7]. Akhawe et al. [2] considered the issues of TLS errors on the web and notices differences with results from OpenSSL. While these studies makes recommendations based on their findings, there is no mechanism for retrofitting changes into existing applications, which we make feasible with CertShim.

Large-scale automated dynamic analysis of SSL usage requires knowledge of application semantics in order to trigger SSL connections, and is therefore difficult on general computing platforms due to the great diversity of languages, code paths, and SSL implementations; however, recent work has made use of the constrained interfaces of mobile platforms to perform large scale analysis. SMV-Hunter leverages knowledge of the `X509TrustManager` interface and Android `WindowManager` to perform user interface automation, triggering SSL connections in hundreds of Android apps to detect MitM vulnerabilities [43]. MalloDroid performs static analysis to identify deviant SSL usage in thousands of apps, but manually audited to confirm vulnerabilities [16].

While the above studies offered recommendations for the general improvement of the SSL ecosystem, such as improved app market testing [16], clarifying SSL APIs [19], or communicating vulnerabilities to developers [7], they were unable to introduce system-wide defenses to SSL vulnerabilities in legacy software. An exception to this is Fahl et al's *Rethinking SSL* work, in which an Android patch is introduced that dramatically improves app security through user interface warnings, device-specific developer options, and forced certificate and hostname validation [17]. While we also introduce a platform-wide defense, our work does not require a manufacturer update, or even administrator privileges, to put to use. Additionally, where pluggable certificate verification is left to future work by Fahl et al., we introduce four such modules, and the ability to use them in tandem through policy-specified consensus votes. Our tool, CertShim, works in a considerably more complex environment than the Android platform, where various SSL implementations need be considered. Both our system and Fahl et al's experience compatibility issues with some programs; however, rather than rely on developers to update their applications, we provide a policy engine that allows for application- or domain-specific certificate handling.

Rather than re-architecting SSL stacks in the OS, other work has invasive strategies to protecting SSL that actually closely mirror the attack behaviors. MYTHIS uses a local MitM network proxy as an SSL security layer on android [11]; by anchoring its security in a single-path network probe, MYTHIS detects *rogue access points* [40], but not attacks near the server or network interior. Huang et al. embed flash scripts in browser code that "phone home" to the server, allowing websites to detect the presence of forged certificates [24]. CertShim also behaves similarly to an attack by hijacking dynamic library calls; however, our solution is a more general one that permits multiple trust models and detects wider classes of attacks.

Various proposals in the literature adopt a similar deployment strategy to CertShim. Provos et al. [36] implement privilege separation (*Privsep*), modifying a small portion of the OpenSSH source code to permit different parts of an application run at different privilege levels. They demonstrate that this approach allowed for interoperability and negligible performance costs. Watson et al. [46] present *Capsicum*, a capability-based sandboxing mechanism for UNIX, through the introduction of a library that replaces basic UNIX operations such as `fork` and `exec`. They present Capsicum-compliant versions of several popular utilities (e.g. tcpdump, gzip), and perform microbenchmarking to demonstrate small overheads on the modified system calls. The DNSSEC-Tools project provided Libval, a shim for the DNS library that facilitated the rapid adoption of DNSSEC [44]. Our work differs from the Libval in that we target multiple SSL implementations used by a greater diversity of programs, override additional functions in the Linux networking stack to track SSL flow context, and employ a modular design that supports multiple verification methods.

8. CONCLUSION

This paper has introduced CertShim, a mechanism that immediately improves the security of Internet communications by interposing on SSL APIs, and even permits the retrofitting of legacy software to support SSL trust enhancements such as Convergence and DANE. Moreover, we have presented a practical mechanism for polling the results of multiple verification methods, further promoting the adoptability of CA alternatives by overcoming their usage limitations. We have also shown that 94% of the SSL usage in Ubuntu's most commonly installed packages are supported by CertShim, and that CertShim secures applica-

tions against some of the most infamous SSL vulnerabilities explored in the literature. This work significantly increases system-wide security of SSL communications in non-browser software, while simultaneously reducing the barriers to evaluating and adopting the myriad alternative proposals to the certificate authority system.

Acknowledgments

We would like to thank Paul van Oorschot, Jeremy Clark, Patrick Traynor, and Boyana Norris for their valuable comments and insight. This work is supported in part by the US National Science Foundation under grant numbers CNS-1118046 and CNS-1254198. Braden Hollembaek was funded in part through an NSF REU supplement.

Availability

Source code for CertShim will be made available from our lab website at `http://sensei.ufl.edu`.

9. ADDITIONAL AUTHORS

Abdulrahman Alkhelaifi, University of Oregon.

10. REFERENCES

[1] libconfig - c/c++ configuration file library. Available: `http://www.hyperrealm.com/libconfig/`.

[2] Akhawe, D., Amann, B., Vallentin, M., and Sommer, R. Here's My Cert, So Trust Me, Maybe? Understanding TLS Errors on the Web. In *Proceedings of the 22nd International World Wide Web Conference (WWW 2013)* (Rio de Janeiro, Brazil, May 2013).

[3] Alicherry, M., and Keromytis, A. D. Doublecheck: Multi-path verification Against Man-in-the-Middle Attacks. In *Computers and Communications, 2009. ISCC 2009. IEEE Symposium on* (2009), IEEE, pp. 557–563.

[4] Amann, B., Sommer, R., Vallentin, M., and Hall, S. No Attack Necessary: The Surprising Dynamics of SSL Trust Relationships. In *ACSAC '13: Proceedings of the 29th Annual Computer Security Applications Conference* (Dec. 2013).

[5] Arkko, J., and Nikander, P. Weak Authentication: How to Authenticate Unknown Principals without Trusted Parties. In *Security Protocols*, B. Christianson, B. Crispo, J. Malcolm, and M. Roe, Eds., vol. 2845 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2004, pp. 5–19.

[6] Bernat, A. R., and Miller, B. P. Anywhere, Any-time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools* (New York, NY, USA, 2011), PASTE '11, ACM, pp. 9–16.

[7] Brubaker, C., Jana, S., Ray, B., Khurshid, S., and Shmatikov, V. Using Frankencerts for Automated Adversarial Testing of Certificate Validation in SSL/TLS Implementations. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy* (San Jose, CA, May 2014).

[8] Buck, B., and Hollingsworth, J. K. An API for Runtime Code Patching. *Int. J. High Perform. Comput. Appl. 14*, 4 (Nov. 2000), 317–329.

[9] Carly, R. Internet Security provider Comodo's CEO Named "Entrepreneur of the Year" by Info Security Products Guide. Available: `http://www.comodo.com/news/press_releases/2011/02/comodo-CEO-entrepreneur-of-the-Year-infosecurity-global-excellence-award.html`, February 2011.

[10] Clark, J., and van Oorschot, P. C. SoK: SSL and HTTPS: Revisiting Past Challenges and Evaluating Certificate Trust Model Enhancements. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy* (San Francisco, CA, May 2013).

[11] Conti, M., Dragoni, N., and Gottardo, S. MITHYS: Mind The Hand You Shake - Protecting Mobile Devices from SSL Usage Vulnerabilities. In *Security and Trust Management*, R. Accorsi and S. Ranise, Eds., vol. 8203 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 65–81.

[12] Ducklin, P. The TURKTRUST SSL Certificate Fiasco – What Really Happened, and What Happens Next? Available: `http://nakedsecurity.sophos.com/2013/01/08/`Available: `the-turktrust-ssl-certificate-fiasco-what-happened-and-what-happens-next/`, January 2013.

[13] Eastlake, D., et al. Transport Layer Security (TLS) Extensions: Extension Definitions.

[14] Eckersley, P. Sovereign Key Cryptography for Internet Domains, 2011.

[15] Edge, J. Mozilla and CNNIC. Available: `http://lwn.net/Articles/372386/`, February 2010.

[16] Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., and Smith, M. Why Eve and Mallory Love Android: An Analysis of Android SSL (in)Security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 50–61.

[17] Fahl, S., Harbach, M., Perl, H., Koetter, M., and Smith, M. Rethinking SSL Development in an Appified World. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2013), CCS '13, ACM, pp. 49–60.

[18] Fisher, D. Microsoft Revokes Trust in Five Diginotar Root Certs. Wired. Available: `http://threatpost.com/microsoft-revokes-trust-five-diginotar-root-certs-mozilla-drops-trust-staat-der-nederland-cert`, September 2011.

[19] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., and Shmatikov, V. The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software. In *Proceedings of the 2012 ACM conference on Computer and communications security* (Raleigh, NC, USA, 2012), CCS '12, ACM, pp. 38–49.

[20] Gibbs, S. Heartbleed Bug: What Do You Actually Need to do to Stay Secure? Available: `http://www.theguardian.com/technology/2014/apr/10/heartbleed-bug-everything-you-need-to-know-to-stay-secure`.

[21] Grigg, I. VeriSign's Conflict of Interest Creates New Threat. *Financial Cryptography 1* (September 2004).

[22] Hickman, K., and Elgamal, T. The SSL Protocol. *Netscape Communications Corp 501* (1995).

[23] Hoffman, P., and Schlyter, J. The DNS-Based Authentication of Named Entities (DANE) Transport Layer Security (TLS) Protocol: TLSA. Tech. rep., RFC 6698, August, 2012.

[24] Huang, L.-S., Rice, A., Ellingsen, E., and Jackson, C. Analyzing Forged SSL Certificates in the Wild.

[25] Laurenzano, M., Tikir, M., Carrington, L., and Snavely, A. PEBIL: Efficient static binary instrumentation for Linux. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on* (March 2010), pp. 175–183.

[26] Laurie, B., Langley, A., and Kasper, E. Certificate Transparency. *Available: ietf. org-Certificate Transparency (06.01. 2013)* (2013).

[27] Liang, J., Jiang, J., Duan, H., Li, K., Wan, T.,

and Wu, J. When HTTPS Meets CDN: A Case of Authentication in Delegated Service.

[28] Marlinspike, M. More tricks for defeating SSL in practice. *Black Hat USA* (2009).

[29] Marlinspike, M. New tricks for defeating SSL in practice. *BlackHat DC* (Feb. 2009).

[30] Marlinspike, M. SSL and the Future of Authenticity. *Black Hat USA* (2011).

[31] Marlinspike, M. Trust Assertions for Certificate Keys.

[32] Mills, E. Comodo: Web Attack Broader Than Initially Thought. CNET. Available: `http://news.cnet.com/8301-27080_3-20048831-245.html?part=rss&tag=feed&subj=InSecurityComplex`, March 2011.

[33] Myers, M. Revocatoin: Options and challenges. In *Financial Cryptography* (1998), Springer, pp. 165–171.

[34] Pennarun, A., Allombert, B., and Reinholdtsen, P. Ubuntu Popularity Contest. Available: `http://popcon.ubuntu.com/`.

[35] Pieter Lexis. SWEDE - A Tool To Create and Verify TLSA (DANE) Records. Available: `https://github.com/pieterlexis/swede`.

[36] Provos, N., Friedl, M., and Honeyman, P. Preventing Privilege Escalation. In *Proceedings of the 12th USENIX Security Symposium* (2003), pp. 231–242.

[37] Psyced.org. Certificate Patrol. Available: `http://patrol.psyced.org/`.

[38] Rivest, R. L. Can We Eliminate Certificate Revocation Lists? In *Financial Cryptography* (1998), Springer, pp. 178–183.

[39] Sandvik, R. Security Vulnerability Found in Cyberoam DPI Devices (CVE-2012-3372). Available: `https://blog.torproject.org/blog/security-vulnerability-found-cyberoam-dpi-devices-cve-2012-3372`, July 2012.

[40] Shetty, S., Song, M., and Ma, L. Rogue Access Point Detection by Analyzing Network Traffic Characteristics. In *Military Communications Conference, 2007. MILCOM 2007. IEEE* (Oct 2007), pp. 1–7.

[41] Singel, R. Law Enforcement Appliance Subverts SSL. Available: `http://www.wired.com/threatlevel/2010/03/packet-forensics`, March 2010.

[42] Soghoian, C., and Stamm, S. Certified Lies: Detecting and Defeating Government Interception Attacks Against SSL. In *Financial Cryptography and Data Security*. Springer, 2012, pp. 250–259.

[43] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings of the 19th Network and Distributed System Security Symposium.* (2014).

[44] SPARTA, Inc. DNSSECTools: DNSSEC Software Libraries and Tools. Available: `http://www.dnssec-tools.org/`.

[45] Vratonjic, N., Freudiger, J., Bindschaedler, V., and Hubaux, J.-P. The Inconvenient Truth About Web Certificates. In *Economics of Information Security and Privacy III*, B. Schneier, Ed. Springer New York, 2013, pp. 79–117.

[46] Watson, R., Anderson, J., Laurie, B., and Kennaway, K. Capsicum: Practical Capabilities for UNIX. In *Proceedings of the 19th USENIX Security Symposium* (2010).

[47] Wendlandt, D., Andersen, D. G., and Perrig, A. Perspectives: Improving SSH-style Host Authentication with Multi-Path Probing. In *USENIX 2008 Annual Technical Conference* (Boston, MA, 2008), ATC'08, pp. 321–334.