2019-01-01

# The Security Layer

Mark Thomas O'Neill
*Brigham Young University*

# The Security Layer

Mark Thomas O'Neill

A dissertation submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Daniel Zappala, Chair
Kent Seamons
Michael Goodrich
Tony Martinez
Ryan Farrell

Department of Computer Science

Brigham Young University

ABSTRACT

The Security Layer

Mark Thomas O'Neill
Department of Computer Science, BYU
Doctor of Philosophy

Transport Layer Security (TLS) is a vital component to the security ecosystem and the most popular security protocol used on the Internet today. Despite the strengths of the protocol, numerous vulnerabilities result from its improper use in practice. Some of these vulnerabilities arise from weaknesses in authentication, from the rigidity of the trusted authority system to the complexities of client certificates. Others result from the misuse of TLS by developers, who misuse complicated TLS libraries, improperly validate server certificates, employ outdated cipher suites, or deploy other features insecurely. To make matters worse, system administrators and users are powerless to fix these issues, and lack the ability to properly control how their own machines communicate securely online.

In this dissertation we argue that the problems described are the result of an improper placement of security responsibilities. We show that by placing TLS services in the operating system, both new and existing applications can be automatically secured, developers can easily use TLS without intimate knowledge of security, and security settings can be controlled by administrators. This is demonstrated through three explorations that provide TLS features through the operating system. First, we describe and assess TrustBase, a service that repairs and strengthens certificate-based authentication for TLS connections. TrustBase uses traffic interception and a policy engine to provide administrators fine-tuned control over the trust decisions made by all applications on their systems. Second, we introduce and evaluate the Secure Socket API (SSA), which provides TLS as an operating system service through the native POSIX socket API. The SSA enables developers to use modern TLS securely, with as little as one line of code, and also allows custom tailoring of security settings by administrators. Finally, we further explore a modern approach to TLS client authentication, leveraging the operating system to provide a generic platform for strong authentication that supports easy deployment of client authentication features and protects user privacy. We conclude with a discussion of the reasons for the success of our efforts, and note avenues for future work that leverage the principles exhibited in this work, both in and beyond TLS.

Keywords: SSL, TLS, man in the middle, certificate, OpenSSL, security, operating system, administrator control, kernel security, policy, certificates, transport layer security, secure sockets layer, developer mistakes, vulnerabilities, client authentication, DANE, OSCP, CRLSet, CRL, Convergence, notary, POSIX, socket API, API

# ACKNOWLEDGMENTS

I'd like to thank my advisor, Daniel Zappala, for allowing me the freedom to pursue my degree in the way and areas I wanted. His immediate, unsolicited, and unending support of me and my ideas was a great source of confidence and encouragement, especially during the most frustrating times.

A small army of peers assisted me in the development of the works described in this dissertation: Benjamin Davis, Dan Bunker, Elham Vaziripour, Jordan Whitehead, Joshua Reynolds, Justin Wu, Luke Dickinson, Mason Coram, Nick Bonner, Scott Heidbrink, Tanner Perdue, Torstein Collett, and Travis Hendershot. Their work has been exemplary, and I'm proud to call every single one of them my friend.

A special thanks goes to Scott Heidbrink, who worked with me from the time this dissertation was a mere fantasy, through its publications, obtaining his master's degree in the process. His dedication, expertise, and wit, did not go unnoticed and will be missed.

Another special thanks goes to my fellow PhD students, Elham Vaziripour and Justin Wu, whose companionship during fun weekend outings and all-nighters in the lab will be forever remembered.

I'd also like to thank those who were involved behind the scenes, including Jen Bonnett, whose help none of us graduate students could graduate without, and all of the babysitters who volunteered their time to watch my son as I worked. Most of all, I'd like to thank my wife, Leah, who sacrificed her time and talents to support me both financially and mentally during these years.

Finally, I want to thank the National Science Foundation, the Department of Homeland Security, and Sandia National Laboratories, for their interest and financial support of this work.

## Table of Contents

## List of Figures

## List of Tables

# Chapter 1

## Introduction

Transport Layer Security (TLS) [86], is the most popular security protocol used on the Internet today.[1] Many network applications use TLS, including email clients, VPN clients, instant messaging services, and all web browsers. When used correctly, TLS provides a variety of security guarantees to a connection between an Internet client and server, including confidentiality, integrity, and authentication. Together, these features allow Internet applications to securely communicate with remote hosts, providing them with cryptographic proof that they are connected to their intended peer and not to an imposter, and the ability to exchange arbitrary information in a tamper-proof manner, without divulging its contents to eavesdroppers. Furthermore, establishing a secure TLS connection only requires the participation from the communicating hosts. Network routers and other machines between the two hosts need not be trusted or directly involved, aside from facilitating data transfer.

Throughout the last decade, the TLS protocol itself has proven to be rather resilient, requiring only minor patches to address weaknesses. These updates typically come in the form of the deprecation of certain cryptographic algorithms or features found to be too weak for modern computational power. However, in practice, TLS is often misused in ways that subject the protocol to attack. Attackers who exploit such misuse can undermine the aforementioned security guarantees, causing applications to divulge sensitive information with unwanted parties or to accept data that have been altered in transit. In a world where

[1] Prior versions of TLS were known as Secure Socket Layer (SSL). Unless otherwise specified, TLS will be used in this document to mean both TLS and SSL.

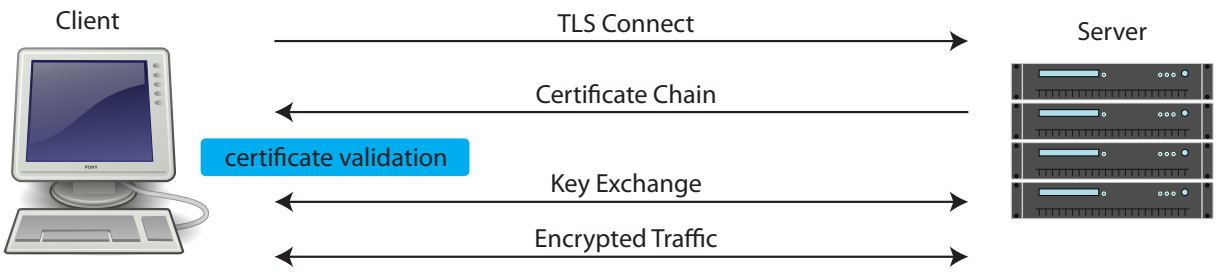



Figure 1.1: High-level example of server authentication under TLS

TLS use is ubiquitous, such weaknesses pose a great threat to personal, organizational, and even national security. These weaknesses fit generally into three areas:

- Authenticating the server to the client
- Securely incorporating TLS in network applications
- Authenticating the client to the server

We now provide a brief overview of each of these areas, and outline the problems they exhibit in practice.

## 1.1 TLS Server Authentication

Under TLS, proper authentication of the server provides the client with assurance that it is connected to the intended server and not an imposter. In this section we outline the mechanism through which this assurance is obtained, and detail areas in which it falls short.

### 1.1.1 Background

Server authentication under TLS is performed before any data is sent between two communicating hosts[2]. A high-level diagram of TLS server authentication is shown in Figure 1.1. While technically optional, most TLS servers offer proof of their identity during TLS session establishment, as no authentication guarantees can be provided without it. To do this, the

[2]Session resumption follows this pattern as well, considering the authentication performed in the original handshake.

server typically sends a chain of digital X.509 certificates to the client. The leaf certificate in the chain is owned by the server itself, and contains the server's public key and other identifying information, such as the server's hostname. Each certificate in the chain is digitally signed by the owner of the next. Entities that can sign certificates are known as certificate authorities (CAs). Clients can validate a certificate chain by cryptographically verifying the chain of signatures back to a CA they trust. Trusted CA certificates are typically shipped within operating systems and browsers. If no invalid signatures or mismatches of other crucial data are found in the certificate chain, the certificate is considered valid by the client. Subsequently, the client and server negotiate a symmetric session key, during which the server will prove ownership of the private key corresponding to the public key contained in its certificate (the details of this process vary, depending on the cipher suites chosen by the two endpoints and the version of TLS used). After a successful session key establishment, all data sent between the two endpoints is then encrypted.

Failure of the client to properly validate a certificate chain sent by a server leaves the connection susceptible to TLS man-in-the-middle (MITM) attacks, wherein attackers impersonate themselves to the client by using their own certificate chain for authentication rather than the original one provided by the server. If the client accepts this forged certificate chain, the attacker can decrypt, read, selectively modify, and re-encrypt traffic between the client and original server. This scenario is depicted in Figure 1.2. Note that an attacker may also choose to not connect to the legitimate server and simply impersonate it to the client without forwarding traffic.

There are numerous ways that an attacker can get a client to accept a different certificate chain from the legitimate one, which are outlined in the succeeding section.

### 1.1.2 Problems in Practice

Three significant problems challenge TLS server authentication. These include insecure validation practices from TLS-using applications, inadequacies of the certificate authority

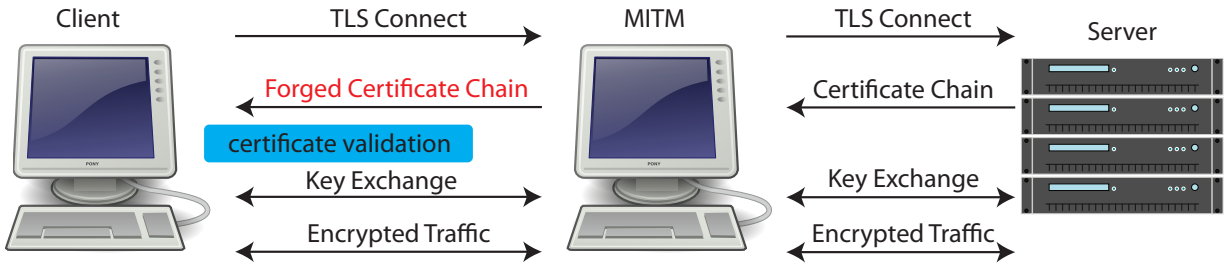



Figure 1.2: High-level example of a MITM attack against TLS using forged certificates

system as a whole, and the adoption difficulties encountered by technologies that attempt to solve these issues.

#### 1.1.2.1 Insecure Validation Practices by Applications

Numerous studies have found that both desktop and mobile applications frequently do not properly validate a server's certificate [14, 31, 32, 35, 68]. Validating a certificate requires multiple complex steps. A certificate chain must be checked to ensure it has valid signatures, proper signing permissions, the correct hostname, unexpired issuance, and other properties. Not all of these properties can be evaluated in isolation; external logs and services must sometimes be queried to determine whether a certificate has been revoked, and applications must further protect themselves by ensuring that certificates were not issued using weak signing algorithms. The omission of any of these checks leaves applications vulnerable to man-in-the-middle attacks. For example, some applications verify the chain of digital signatures in a certificate chain but fail to ensure that the hostname in the leaf certificate matches the hostname of the intended server. In such cases, any attacker with a valid certificate for one hostname can use his certificate to impersonate a host of his choosing to the application.

Application developers alone are not to blame for validation problems. Common security libraries and middleware solutions, upon which applications rely, are often too complex and poorly-designed for developers to use properly. For example, under OpenSSL, certificate hostname validation does not occur automatically. Calling the function `SSL_set_verify` (or

SSL_CTX_set_verify) with a setting of SSL_VERIFY_PEER does not fix this issue, as it only enables a default validation function with some minimal checks, and the trust store used should be set previously using SSL_CTX_load_verify_locations. A developer can choose to supply a custom callback to the SSL_set_verify function known as a "verify callback", but this function can be confusing as it is invoked once per certificate in the chain, forcing developers to write extra code to ensure they are checking the hostname of the leaf certificate. The best practice is actually to supply a custom validation function using a different OpenSSL endpoint, SSL_CTX_set_cert_verify_callback, not to be confused with the other "verify callback". Using this callback developers have access to the whole certificate chain in a single invocation, and can provide their additional validation. However, using this feature overrides the aforementioned minimal validation, and thus it must be invoked manually using X509_verify_cert. At this point, developers are free to perform hostname validation, but prior to OpenSSL version 1.1, there was no OpenSSL function to provide hostname validation, so developers targeting version 1.0 and below have to develop it themselves.

#### 1.1.2.2 Certificate Authority System Drawbacks

The CA system is vulnerable to being hijacked even when applications are implemented correctly. This is largely due to the fact that most CAs are able to sign certificates for any hostname, reducing the strength of the CA system to that of the weakest CA [25]. This weakness was exploited in 2011 when DigiNotar's servers were hacked and more than 500 certificates were fabricated by the intruder, including a certificate for Gmail that allowed the intruder to access stored email for 300,000 Iranians [47]. This happened despite the fact that Gmail does not use DigiNotar to sign its certificates.

This problem is exacerbated by CAs that do not follow best security practices [22, 55] and governmental ownership and access to CAs [26, 84]. In these cases, impersonating entities can easily obtain legitimate certificates from legitimate authorities, undermining the purpose of the entire system. For example, Symantec's certificate authority business came

into conflict with Google after it was discovered that they had issued thousands of improper certificates and allowed numerous subsidiary organizations to issue certificates without proper oversight [62]. As a governmental example, the China Internet Network Information Center (CNNIC) is a certificate authority founded by the Chinese government, under the guise of a nonprofit entity. In 2015, this CA was trusted by nearly all desktop and mobile systems, and was found to issue a signing certificate to Mideast Communication Systems, who in turn used it to generate several certificates for Google domain names [36]. Google noticed this offense and subsequently removed CNNIC from the trust store of all their products. However, other CAs that are influenced or operated by governments are still trusted by many devices.

Related to these issues is the presence of TLS proxies, sometimes implemented using middleboxes, that dynamically generate forged certificates and essentially perform a MITM attack on clients for firewall purposes. Corporations may deploy a TLS proxy to prevent leakage of company secrets through secure channels, provide real-time malware screening, and block malicious websites. Home users may use personal firewall software equipped with TLS proxy functionality to safeguard themselves from malware. Some device manufacturers, such as Nokia, have also leveraged their position to install their own CA certificates on devices, unbeknownst to their users [60]. In these cases, the forged certificates are trusted because the company or firewall solution installs its own CA certificate in the trust store of the device. This practice is wrought with controversy, given its privacy implications. While the protective aspects of TLS proxies are generally supported by users, an overwhelming majority of them also wish to be informed when such proxies are active [64, 79]. Unfortunately, many of these proxies have also been found to downgrade security, sometimes in critical ways [24]. Malware has also been found to leverage this same technique, making distinguishing between benevolent and malicious TLS proxies a difficult task [65].

#### 1.1.2.3 Adoption of Alternative and Strengthening Technologies

In response to the brittle nature of the CA system and its misuse by applications, numerous technologies have been introduced which attempt to strengthen or replace it entirely. An example of a strengthening technology is Certificate Revocation Lists (CRLs) [16], which provide clients with the ability to ensure that a certificate is not revoked before accepting it. Convergence [55], on the other hand, allows TLS clients to completely abandon the CA system in favor of validating certificates via consensus, from a set of trusted certificate monitors. Unfortunately, there is no widely-adopted platform assisting the deployment of improvements to the CA system, meaning researchers and developers have to individually modify applications to make advances, which severely limits their deployment. New authentication solutions oftentimes repeat much of the work of their predecessors as they all require similar exported data to operate. Browser vendors, while generally better at adopting these new technologies compared to other applications, sometimes adopt different technologies than their competition. For example, Mozilla Firefox only supports revocation via the Online Certificate Status Protocol whereas Google Chrome supports it through CRLs.

## 1.2 Using TLS in Applications

In this section we turn our attention to the second problematic area of TLS: incorporating TLS into network applications. In similar fashion to our server authentication discussion, we briefly introduce developer interactions with TLS and then discuss challenges.

### 1.2.1 Background

Typically C/C++ programs rely on security libraries such as OpenSSL or GnuTLS for TLS support. These libraries are often wrapped by higher-level languages to provide TLS support, such as with Python's `ssl` module. In using these libraries, developers specify what version of TLS their programs should use, what cipher suites (e.g., encryption and hashing algorithms) should be employed, how certificates should be validated, etc. Even without considering the

aforementioned issues with certificate validation, incorrect TLS usage can have dire effects on security. Developers incorporating TLS need to ensure that they are not using it in known vulnerable ways. For example, using older modes of TLS renegotiation allows attackers to inject their own traffic into the TLS session. Weak algorithm usage can also be problematic. Under TLS, the client and server negotiate a "cipher suite" that dictates what algorithms they use for various encryption and authentication tasks. As computational power increases and more attacks against algorithms are found, developers need to ensure they are not continuing to employ weak cipher suites.

### 1.2.2 Problems in Practice

Security libraries are known to be hard to use properly, as exemplified in Section 1.1.2.1. Our work finds that the `libssl` component of the OpenSSL 1.0 library alone exports 504 functions and macros for use by TLS-implementing applications. This problem is likely to persist, as the unreleased OpenSSL 1.1.1 has increased this number substantially. OpenSSL and other TLS APIs have been criticized for their complexity [35, 38] and, anecdotally, our own explorations find many functions within `libssl` that have non-intuitive semantics, confusing names, or little-to-no use in applications.

The amount of developer time required to use TLS can also be problematic. As a demonstration, we developed a simple web client in C that issues a request for a given homepage, and displays the response. Sixty lines of code were required for this functionality using the POSIX socket API. TLS 1.2 support was then added to the client using OpenSSL 1.0.2, with hostname validation code from the cURL library. This resulted in the addition of another 266 lines of code, for a total of 326. Contrasted with precedent for other protocols, the magnitude of this increase is unusual. For example, network programmers do not have to add any code to interact with complex TCP or UDP libraries to use those protocols; they simply request a given protocol and use the resulting socket for application-layer interactions. Given the relative low population of security experts to that of the greater application developer

population, it is impractical to assume that each application will properly use existing security libraries. Furthermore, developers tend to focus on features that make their applications unique and regularly defer or omit security, even when they are aware of security and privacy needs [52].

A related issue is that placing the burden of TLS integration on application developers also fragments TLS configurations, making it more difficult to phase out old versions and ciphers as general vulnerabilities are found. For example, despite the deprecation of TLS version 1.0, our analysis of 410 Linux software packages revealed that TLS 1.0 was still explicitly supported by 195, followed by another 83 that still supported the SSL 3.0, its vulnerable predecessor.

A final drawback is that all popular TLS libraries share an address space with the applications that use them. Thus vulnerabilities in the application may also put sensitive TLS session data at risk, such as master keys for the session or private keys corresponding to certificates used.

## 1.3 TLS Client Authentication

The final area of focus for TLS difficulties is the authentication of clients to the server. In this section we will provide an overview of this functionality and then discuss its usability and deployment difficulties in practice.

### 1.3.1 Background

Mutual authentication between the client and server is supported by all versions of TLS. The process of client certificate validation by the server is very similar to the process followed by clients during server certificate validation. Client certificates are of the same format as their server counterparts, and validation is performed in nearly the same way. Instead of hostname identifiers, client certificates usually contain names, usernames, or email addresses. Servers may request that the client provide a certificate to prove its identity during the

TLS handshake, before any data is exchanged. If client authentication is not required at the beginning of a connection, a secure TLS renegotiation may be used when it is later required, which triggers a new handshake, allowing the client to then provide a certificate. TLS 1.3 supports this more naturally, deprecating renegotiation entirely and providing a native "post-handshake authentication" mechanism that allows servers to request client authentication at any time during a connection. When requesting client authentication, the server can provide a list of CAs that it trusts, allowing the client to select an appropriate certificate to use. In response, the client will provide the proper certificate and also prove its ownership of that certificate by signing a transcript of the messages sent between the two hosts up to that point, with the corresponding private key. The client can also opt to deny the authentication request and instead send an empty certificate message in response. In either case, the server then decides whether and how to continue the active TLS session.

TLS client authentication has two important advantages in the context of our discussion. First, it raises the bar for MITM attacks. In a traditional MITM attack using a TLS proxy, attackers have to trick the client application into accepting a forged certificate. Under client authentication, attackers must also trick the server to accept a forged certificate. This is because the client signs a transcript of the previous handshake messages sent between the sever and client, using the private key of its certificate. Since the transcript includes the certificate of the server and some unique connection establishment data, this signature binds the authentication to a single TLS connection. Second, client authentication does not divulge any password-equivalent data to the server. Since certificate validation relies on public key cryptography, attackers hoping to masquerade as the server through phishing or other types of impersonation attacks are unable to obtain any login credentials (e.g., usernames and passwords), even if the client erroneously establishes a connection to the malicious server. In this fashion TLS mutual authentication, when performed properly, provides a natural defense against password theft, phishing, and similar attacks; login credentials cannot be stolen because they are never transmitted.

### 1.3.2 Problems in Practice

In contrast to the problems found in TLS server certificate validation and employing TLS in applications, the main problem surrounding TLS client authentication is its lack of use, rather than its misuse. Another two problems stem from privacy leakage and usability difficulties. We now detail each of these.

#### 1.3.2.1 Lack of Use

While TLS client authentication is not widely used, the need for some means of client authentication remains. As a result, this authentication, both with browsers and other Internet applications, is typically performed using passwords. While a TLS extension implementing a Secure Remote Password protocol does exist (TLS-SRP [86]), most applications tend to use plain passwords at the application layer. For example, a user browsing Amazon.com with a web browser is asked to log in through a form on a webpage, after a TLS connection has already been established. Thus server authentication is performed by with TLS, and client authentication is performed by the web server using code at the application layer, with HTTP. This practice necessitates that applications implement and manage their own login mechanisms, instead of relying on a TLS implementation for authentication (which would be preferable, if security libraries were easy to use). In addition, the use of passwords in this fashion causes three important problems:

1. **Passwords must be transmitted:** Both parties must know a password for it to be useful. When proving knowledge of the password for authentication as described, clients must transmit the password to the server. If the server is properly authenticated using TLS beforehand, this is not a critical issue. However, improper server authentication or fooling users via phishing can cause users to connect to a malicious server posing as a legitimate one. In this case, users provide their plaintext credentials directly to the attacker, in turn allowing the attacker to impersonate the client to the real server.

2. **Passwords must be stored:** Servers must store password data for comparison during the login attempts from users. Best practice for storing passwords includes salting them with a random value and then hashing them with a strong hash algorithm. Unfortunately, not all servers follow these guidelines, including large organizations with millions of users, which leaves login credentials vulnerable when the server is compromised by attackers. We offer two examples. In 2012, LinkedIn notified the public that a compromise of their password database leaked 6.5 million account credentials [83], which had not been salted and used the weak SHA-1 algorithm for hashing. Later in 2016, LinkedIn discovered that the actual number of compromised accounts was 167 million [82] and credentials were for sale in digital black markets. Adobe was compromised in 2013, leaking 38 million account credentials in the hands to the public, wherein passwords were encrypted (not hashed) with the known-broken 3DES algorithm, not salted, and stored with corresponding password hints in plaintext [21, 75].
3. **Passwords are reused:** Since so many Internet hosts authenticate clients using passwords, users are often overburdened with remembering a variety of strong passwords, opting instead to reuse their passwords across services [18]. This exacerbates the previous two problems, allowing attackers to target weaker servers to steal credentials for stronger ones.

#### 1.3.2.2 Privacy Leaks

Under TLS 1.2 and below, client certificates are sent to the server before an encrypted tunnel is formed, leaving information that can identify a client in the hands of eavesdroppers. This has been cited as a reason to avoid TLS client certificates [8] and others have demonstrated the ability to precisely track online users based on this vulnerability [88]. Fortunately, under the recently-finalized TLS 1.3, much of the handshake is now encrypted, including client certificates, mitigating this privacy concern.

#### 1.3.2.3 Certificate Usability

A final problem with TLS client authentication is the usability of certificates themselves. First, a certificate is only useful if its owner has access to the corresponding private key. Since these keys are impractical to memorize, they are not implicitly portable like passwords. The loss or theft of a private key can likewise be difficult to handle, as there is no standard analog to password reset for keys. Lastly, certificate enrollment for users can be a cumbersome task, and subsequent use of client certificates can also be difficult for users and server administrators. User interfaces provided by browsers, and configuration directives offered by webservers like Apache, are often complicated and largely to blame for this issue. Parsovs provides an excellent overview of these usability and security issues, and interested readers are referred to that work for more details [71].

## 1.4 Synthesis

We have covered problems in three areas of TLS use in practice: server authentication, use of TLS in applications, and client authentication. Taken as a whole, we find a commonality among all areas: security responsibilities are misplaced. Specifically, the responsibilities of network security policy and practice is scattered in an ad-hoc manner, being placed upon entities that are ill-equipped for those tasks. We articulate this in the context of each problem area:

- **Server Authentication:** The responsibility of proper certificate validation is placed on application developers, who have been repeatedly shown to make critical errors in this process, and who consistently prioritize functionality over security. To make matters worse, the certificate authority system is wrought with its own problems which provide remote, untrustworthy organizations with the ability to override local authentication decisions, in a covert manner. Finally, there is no central mechanism upon which applications can rely to allow users to distance themselves from the CA system or

enhance it, using alternative technologies. Users must simply hope that applications developers adopt technologies and policies consistent with their needs.

- **TLS Use by Applications:** The responsibility of TLS use is also placed on application developers, who are expected to keep up with modern security trends and implement network security functionality despite a lack of expertise. While libraries like OpenSSL provide a wealth of cryptographic utility, they were not built with the separation of security policy and application functionality in mind, publishing hundreds of developer-facing API endpoints that are frequently misused. As a result developers are dealt an unfair hand, required to implement network security, provided with a cumbersome toolset, and expected to divine user security needs. Meanwhile, administrators seeking to control what versions of TLS are used on their machines, with corresponding cipher suites, extensions, and other configuration options, are out of luck.
- **Client Authentication:** The prevalence of password-based application-layer login systems forces users to trust remote servers with their credentials, and to store those credentials securely, despite evidence that even large technology companies do not do so. These credentials are vulnerable to phishing attacks and server data breaches, which occur regularly and can compromise hundreds of millions of accounts at once. Password reuse exacerbates this problem, placing additional cognitive load on users to remember unique passwords for each service they use.

This misplacement of responsibility illustrates the vague nature of the upper layers of various models of the networking protocol stack. The Open Systems Interconnection (OSI) model, shown in Table 1.1, has been used for decades for the conceptual separations of responsibilities of various networking functionality. While most depictions are consistent regarding the roles and example protocols of the lower layers, layers 5 - 7 are far less well-defined. SSL and TLS are sometimes shown to be part of the sixth or fifth layer, but this placement begs an important question: if TLS is below the application layer, then why are applications expected to implement it? The truth is that TLS has no place in the OSI model.

| Layer | Name | Examples |
|---|---|---|
| 7 | Application | HTTP, FTP, SMTP |
| 6 | Presentation | MIME, XDR |
| 5 | Session | Sockets (session establishment) |
| 4 | Transport | TCP, UDP, SCTP |
| 3 | Network | IP, IPSec, ICMP |
| 2 | Data-link | Ethernet, PPP |
| 1 | Physical | Optical Fiber, Coaxial Cable |

Table 1.1: A typical representation of the OSI model. Depictions often differ in their descriptions and example protocols for layers 5 - 7.

In fact, the original OSI paper has no mention of the word "security" [95], let alone a layer for it. The more modern TCP/IP model (also called the Internet model) condenses the OSI layers into a mere four: Link, Internet, Transport, and Application. Most descriptions of this model make mention of TLS at the Application layer, if it is mentioned at all. While this placement is more accurate in terms of the status quo, it does little to prescribe a proper delineation of security responsibilities from those of the rest of the application. Other, less popular models exist as well, but none include an explicit security layer. Overall, current models reflect the sorry state of network security: an orphan over which no one claims clear responsibility.

### 1.5 Solution: A Well-formed Security Layer

In this dissertation, we explore the creation of a true security layer for the model, placing this layer between the fourth and fifth layers of the OSI model and between Transport and Application layers for the TCP/IP model. More importantly, we demonstrate how this placement of security solves the problems with TLS discussed so far. Many of the benefits of this placement stem from an important characteristic of the fifth layer: it straddles the boundary between userspace and kernelspace. More specifically, applications use sockets as their basic interface to network functionality, and most operating systems provide the

functionality below the socket layer. Thus, a layer between the fourth and fifth OSI layers can leverage the power of the operating system to control security behavior system-wide, and also expose a simple API to application developers to use this functionality.

Placing network security responsibilities in the operating system is a natural fit for three reasons:

- **Trust:** Operating systems are the most trusted and privileged software on a system. A user who can't trust his operating system should likely not be using the machine running it in the first place, as no security guarantees of any kind can be granted anyway[3].
- **Control:** Applications cannot access network functionality except through the system call layer, relying on the operating system for this functionality. As a result, the operating system is in a perfect position to provide centralized network security functionality for all applications, and enforce that local security policies are met.
- **Configuration:** Operating systems respond to administrator configuration. Users have differing security requirements, and until now have had to rely on application developers to meet these needs. System administrators have a much better understanding of their own needs, and can use the operating system to enforce these system-wide. For regular users, OS vendors assume the role of a local administrator, supplying default configurations and adjusting them as needed with OS updates.

We explore the benefits and drawbacks of our approach by making TLS an operating system service. We do this in three phases, with each one tackling one of the TLS problem areas previously identified. These phases form the basis of Chapters and are outlined below.

1. **TrustBase:** We address problems in TLS server authentication by creating TrustBase, an architecture to repair and strengthen certificate-based authentication. TrustBase is placed in the operating system and monitors all network flows for TLS connections

[3]We note possible exceptions when using technologies like Intel's SGX and ARM's TrustZone

established by applications. The certificates from these connections are independently validated by a policy engine that obeys local security policy. This enforces safe defaults and administrator preferences for all authentication decisions, for all applications, and regardless of any programmer mistakes. The policy engine is also pluggable, allowing system-wide deployment of new technologies that strengthen the CA system or replace it entirely.

2. **The Secure Socket API:** We address the problems in application use of TLS by introducing the Secure Socket API (SSA), a TLS API implemented as an operating system service. This API reduces the amount of code required to build TLS applications by thousands of lines and allows insecure applications to adopt TLS with as little as one line of code. The SSA itself is built within the confines of the POSIX socket API, allowing network developers to use an API already familiar to them for TLS functionality, using TLS as if it were a built-in protocol such as TCP or UDP. We also provide SSA support to three other languages in addition to C/C++, and provide tools to force applications written against other APIs to use the SSA dynamically. OS vendors and administrators are free to configure the SSA as they see fit, dictating their choice of TLS parameter values on a global and per-app basis.

3. **Revisitation of Client Authentication:** We address the problems in TLS client authentication by leveraging the latest TLS standard, version 1.3, and SSA enhancements. Using these tools we provide an easy-to-use, privacy-preserving and phishing-resistant method of replacing passwords with TLS client certificates. Using the SSA, server developers can implement a login system with two lines of code, independent of any application-layer functionality (such as HTTP WebAuth). In addition to authentication, servers can request various types of client authorization, such as credit card expenditures. Clientside, the SSA manages login for users in the operating system, transparently handling all the complexities of certificates and keys. Users can employ authentication

devices, such as mobile phones, to keep their private keys isolated from their computers and mobile.

We now explore network security as an operating system service.

# Chapter 2

# TrustBase

In this chapter we explore the validation of TLS server certificates in the operating system, granting administrators full control over the trust decisions made by their machines, and allowing applications to rely on a centralized service for server authentication. The text of this chapter is from the following published article:

> O'Neill, M., Heidbrink, S., Ruoti, S., Whitehead, J., Bunker, D., Dickinson, L., Hendershot, T., Reynolds, J., Seamons, K., & Zappala, D. *TrustBase: An Architecture to Repair and Strengthen Certificate-based Authentication.* In proceedings of USENIX Security Symposium, 2017.

## 2.1 Abstract

The current state of certificate-based authentication is messy, with broken authentication in applications and proxies, along with serious flaws in the CA system. To solve these problems, we design TrustBase, an architecture that provides certificate-based authentication as an operating system service, with system administrator control over authentication policy. TrustBase transparently enforces best practices for certificate validation on all applications, while also providing a variety of authentication services to strengthen the CA system. We describe a research prototype of TrustBase for Linux, which uses a loadable kernel module to intercept traffic in the socket layer, then consults a userspace policy engine to evaluate certificate validity using a variety of plugins. We evaluate the security of TrustBase, including a threat analysis, application coverage, and hardening of the Linux prototype. We also

describe prototypes of TrustBase for Android and Windows, illustrating the generality of our approach. We show that TrustBase has negligible overhead and universal compatibility with applications. We demonstrate its utility by describing eight authentication services that extend CA hardening to all applications.

## 2.2 Introduction

Server authentication on the Internet currently relies on the certificate authority (CA) system to provide assurance that the client is connected to a legitimate server and not one controlled by an attacker. Unfortunately, certificate validation is challenged by significant problems. First, applications frequently do not properly validate the server's certificate [14, 31, 35, 68]. This is caused by failure to use validation functions, incorrect usage of libraries, and also developers who disable validation during development and forget to enable it upon release. Second, TLS interception, used by numerous firewall appliances and software (as well as malware), compromises the integrity of end-to-end encryption [44, 65], with many firewalls having significant implementation bugs that break authentication [19, 24]. Third, the CA system itself is vulnerable to being hijacked even when applications and proxies are implemented correctly. This is largely due to the fact that most CAs are able to sign certificates for any host, reducing the strength of the CA system to that of the weakest CA [25]. This weakness was exploited in the 2011 DigiNotar hack [47], and is exacerbated by CAs that do not follow best practices [22, 55] and by governmental ownership and access to CAs [26, 84].

Due to these problems, there are a number of recent proposals to improve or replace the current CA trust model. These include multi-path probing [2, 41, 55, 89] or other systems that vouch for the authenticity of a certificate [4, 5, 29], DNS-based authentication [39], certificate pinning [30, 56], and audit logs [28, 48, 49, 80]. Unfortunately, the majority of applications have not yet integrated these improvements. Even relatively simple fixes, such as certificate revocation, are beset with problems [51]. The result is that there is no de facto

standard regarding where and how certificate validation should occur, and it is currently spread between applications, TLS libraries, and interception proxies [24].

Several projects have tried to address these issues, fixing broken authentication in existing applications, while providing a means to deploy improved authentication services. Primary among these is CertShim, which uses the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries [10]. However, this approach does not provide universal coverage of all existing applications, does not provide administrators singular control over certificate authentication practices, does not protect against several important attacks, and has significant maintenance issues. Fahl takes a different approach that rewrites the library used for authentication by Android applications [32], while also including pluggable authentication modules. This approach is well-suited for Android because all applications are written in Java, but it is difficult to extend this approach to operating systems that provide more general programming language support.

In this paper, we explore a different avenue for fixing these problems by centralizing authentication as an operating system (OS) service and giving system administrators and OS vendors control over authentication policy. These two motivating principles result in TrustBase, an architecture for certificate authentication that secures existing applications, provides simple deployment of improved authentication services, and overcomes shortcomings of previous approaches. TrustBase provides universal coverage of existing applications, supports both direct and opportunistic TLS[1], is hardened against unprivileged local adversaries, is supported on both mobile and desktop operating systems, and has negligible overhead.

To centralize authentication as an operating system service, TrustBase uses a combination of traffic interception to harden certificate validation for existing applications and a validation API to simplify authentication for new or modified applications. TrustBase intercepts network traffic between the socket layer and the transport layer, where it detects the initiation of TLS connections, extracts handshake information, validates the server's

[1]Opportunistic TLS is TLS initiated via an optional upgrade from a plaintext protocol.

certificate using a variety of configurable authentication services, and then allows or blocks the connection based on the results of this additional validation. This allows TrustBase to harden certificate validation in an application-agnostic fashion, irrespective of what TLS library is employed. TrustBase also includes a simple certificate validation API that applications call directly, which extends authentication services to new or modified applications, while also providing compatibility with TLS 1.3.

To provide system administrator control, TrustBase provides a policy engine that enables an administrator to choose how certificate authentication is performed on the host, with a variety of authentication services that can be used to harden the CA system. The checks performed by authentication services are complementary to any existing certificate validation performed by applications. This approach both protects against insecure applications and transparently enables existing applications to be strengthened against failures of the CA system. For example, a browser that validates the extended validation (EV) certificate of a bank is doing the best it currently can, but it is still vulnerable to a compromised CA, allowing a man-in-the-middle (MITM) to present fake but valid certificates. One possible use of TrustBase is to configure the use of notaries that check whether hosts across the Internet are exposed to the same certificate for the bank.[2]

TrustBase enables system administrators and OS vendors to enforce a number of policies regarding TLS. For example, an administrator could require revocation status checking, disallow weak cipher suites, or mandate that Certificate Transparency be used to protect against active man-in-the-middle (MITM) attacks. An OS vendor could ship TrustBase with strong default protections against broken applications, such as enforcing best practices for validating a certificate chain, requiring hostname validation, and pinning certificates for the most popular web sites and applications. As TLS becomes more widespread, TrustBase could easily be extended to provide the capability to report on the use of all applications that do

[2]Keys for notaries can be pinned in advance, so they are not vulnerable to the MITM.

*not* use TLS, so that an organization could better manage or even block insecure applications. All of these improvements can be made without requiring user interaction or configuration.

Our contributions include:

- **An architecture for certificate validation that prioritizes operating system centralization and system administrator control:** TrustBase offers standard certificate validation procedures and optionally adds additional authentication services, both of which are enforced by the operating system and controlled by the administrator or OS vendor. This repairs broken validation for poorly-written applications and can strengthen the validation done by all applications. TrustBase provides a policy engine that enables an administrator to use policies that define how multiple authentication services cooperate, for example using unanimous consent or threshold voting.
- **A research prototype of TrustBase:** We develop a loadable kernel module that provides general traffic interception and TLS handling for Linux. This module communicates via the Netlink API to the policy engine residing in user space for parsing and validation of certificates. We describe how this same architecture can be implemented on other operating systems and give details of our current Android and Windows versions. We provide source code and developer documentation for our prototypes, with licensing for both commercial and non-commercial purposes.
- **A security analysis of TrustBase:** We provide a security analysis of TrustBase, including its centralization, application coverage, and the hardening we have done on the Linux implementation. We describe a threat analysis and demonstrate how TrustBase can thwart attacks that include a hacked CA, a subverted local root store, and a STARTTLS downgrade attack. We also demonstrate the ability of TrustBase to fix applications that do not validate hostnames or skip certificate validation altogether.
- **An evaluation of TrustBase:** We evaluate the TrustBase prototype for performance, compatibility, and utility. (1) We show that TrustBase has negligible performance

overhead, with no measurable impact on latency. (2) We demonstrate that TrustBase enforces correct certificate validation on all popular Linux libraries and tools and on the most popular Android applications. (3) We describe eight authentication services that we have developed and report on how simple and straightforward it was to develop these services for TrustBase.

## 2.3 Related Work

Three systems aim to tackle similar problems as TrustBase.

Fahl et al. proposed a new framework for Android applications that would help developers correctly use TLS [32]. Their work follows a similar principle to ours—instead of letting developers implement their own certificate validation code, validation is a service, and it incorporates a pluggable framework for authentication schemes. Fahl's approach is well-suited to mobile operating systems such as Android, where all applications are written in Java, but it is difficult to extend this approach to operating systems that provide more general programming language support.

Another Android system, MITHYS, was developed to protect Android applications from MITM attacks [15]. It first attempts to MITM applications that establish TLS connections and, if successful, continues using the MITM and provides certificate validation using a notary service hosted in the cloud. MITHYS only works for HTTPS traffic, adds significant delays to all TLS connections that it protects (one to ten seconds), and only supports the current CA system.

The most closely related system to TrustBase is CertShim [10]. Like TrustBase, CertShim is an attempt to immediately fix TLS problems in existing apps and also support new authentication services. CertShim works by utilizing the `LD_PRELOAD` environment variable to replace functions in dynamically-loaded security libraries with their own wrappers for those functions. This method has an advantage over TrustBase in that CertShim does not need to perform double validation for cases where an application is already performing

certificate validation correctly. Because TrustBase uses traffic interception to enforce proper certificate validation, its checks are in addition to what applications may already do (either correctly or incorrectly). In addition, CertShim's wrapping of validation functions means that it can more easily override the CA system in the case where administrators want an application to accept alternative certificates, though this will only work with applications that CertShim supports and that do not validate against hard-coded certificates or keys.

TrustBase has advantages that set it apart from CertShim in several notable ways:

(1) **Coverage.** TrustBase intercepts all secure traffic and thus can independently validate certificates for all applications, regardless of what library they used, how they were compiled, what user ran them, or how they were spawned. CertShim does not support browsers, and it cannot perform validation for applications in all scenarios. For example, applications using custom or unsupported security libraries (e.g., BoringSSL, NSS, MatrixSSL, more-recent GnuTLS, etc.), applications statically linked with any security library, and applications spawned without being passed CertShim's path in the `LD_PRELOAD` environment variable (e.g., spawned by `execv` or spawned by a user without that environment setting) will not have their certificates validated by CertShim.

(2) **Maintenance.** TrustBase only needs to maintain compatibility with the TLS specification and the signatures of high-level functions of TCP in the Linux kernel. As a datapoint, the latter has had only two minor changes since Linux 2.2 (released 1999)—one change was to add a parameter, the other was to remove it. In contrast, CertShim relies on data structures internal to the security libraries it supports, and libraries change their internals with surprising frequency. The current versions of PolarSSL (now mbed TLS) and GnuTLS were no longer compatible with CertShim, one year after its release.

(3) **Administrator Control.** TrustBase ensures that only system administrators can load, unload, bypass, or modify its functionality, so that every secure application is subject to its configured policies. With CertShim, guest users and applications can easily opt out of its security policies by removing CertShim from their `LD_PRELOAD` environment

variable, and developers can bypass CertShim by statically-linking with security libraries, using an unsupported TLS library, or spawning child processes without CertShim in their environment.

(4) **Local Adversary Protection.** TrustBase uses a trust model that protects against a local adversary, wherein a nonprivileged, local, malicious application attempts to bypass or alter certificate validation. Recent studies of TLS MITM behavior suggest that local malware acting as a MITM is more prevalent than remote MITM attackers [44, 65]. TrustBase protects against this case by using a protected Netlink protocol, privileged policy engine, protected files, and kernel module that cannot be removed by a nonprivileged user. CertShim's attack model does not address this case. In fact, malware uses the same `LD_PRELOAD` mechanism [50].

(5) **Opportunistic TLS Enforcement.** TrustBase can enforce the use of TLS in plaintext protocols that optionally allow upgrades to TLS, such as STARTTLS, significantly reducing the attack surface for downgrade attacks. Since CertShim hooks into TLS library calls, it cannot be invoked if no calls occur.

## 2.4 TrustBase

TrustBase is motivated by the need to fix broken authentication in applications and strengthen the CA system, using the two motivating principles that authentication should be centralized in the operating system and system administrators should be in control of authentication policies on their machines. In this section, we discuss the threat model, design goals, and architecture of the system.

### 2.4.1 Threat Model

In our threat model, an active attacker attempts to impersonate a remote host by providing a fake certificate. Our attacker includes remote hosts as well as MITM attackers located

anywhere along the path to a remote host. The goal of the attacker is to establish a secure connection with the client.

The application under attack may accept the fake certificate for the following reasons:

- The application employs incorrect certificate validation procedures (e.g., limited or no validation) and the attacker exploits his knowledge of this to trick the application into accepting his fake certificate.
- The attacker or malware managed to place a rogue certificate authority into the user's root store (or another trust store used by the application) so that he has become a trusted certificate authority. The fake certificate authority's private key was then used to generate the fake certificate used in the attack.
- Non-privileged malware has altered or hooked security libraries the application uses to force acceptance of fake certificates (e.g., via malicious OpenSSL hooks and LD_PRELOAD).
- A legitimate certificate authority was compromised or coerced into issuing the fake certificate to the attacker.

Local attackers (malware) with root privilege are outside the scope of our threat model. In addition, we consider only certificates from TLS connections directly made from the local system to a designated host, and not those that may be present in streams higher up in the OSI stack or indirectly from other hosts or proxies via protocols like onion routing.

### 2.4.2 Design Goals

The design goals for TrustBase are: **(1) Secure existing applications.** TrustBase should override incorrect or absent validation of certificates received via TLS in current applications. **(2) Strengthen the CA system.** TrustBase should provide simple deployment of authentication services that strengthen the validation provided by the CA system. **(3) Full application coverage.** All incoming certificates should be validated by TrustBase, including those provided to both existing applications and future applications. However,

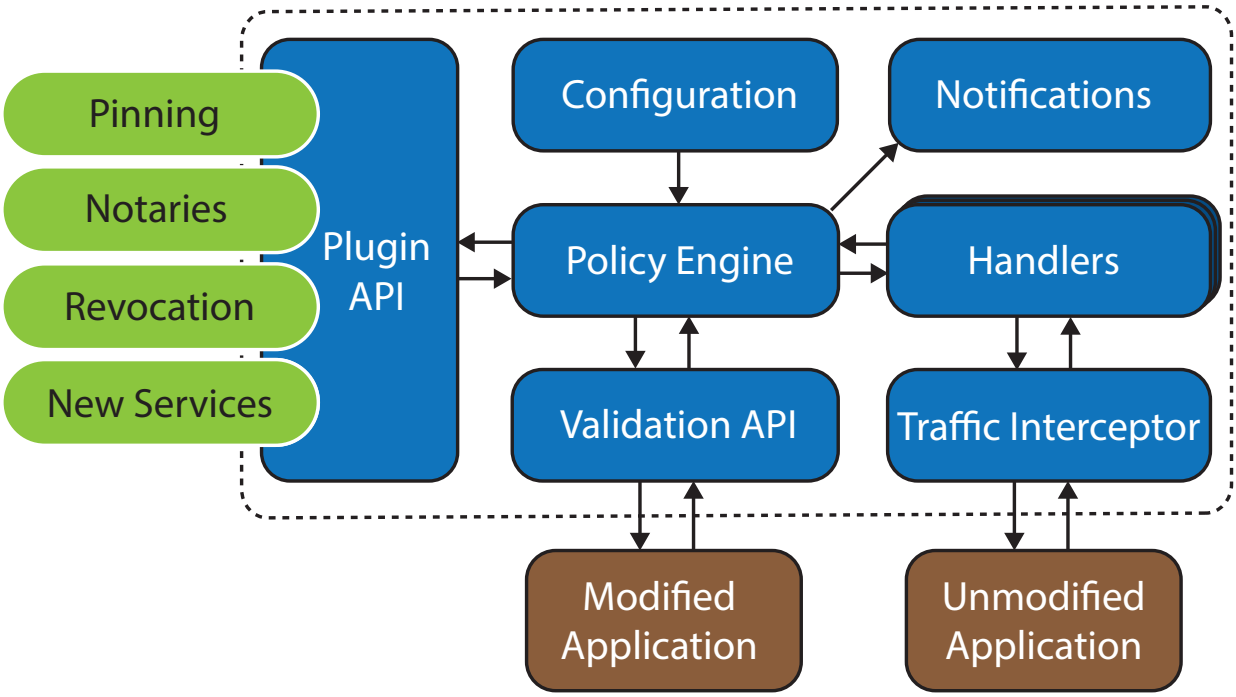



Figure 2.1: TrustBase architecture overview

this does not include certificates from connections not made directly by the system, such as certificates delivered through onion routing. **(4) Universal deployment.** The TrustBase architecture should be designed to work on any major operating system, including both desktop and mobile platforms. **(5) Negligible overhead.** TrustBase should have negligible performance overhead. This includes ensuring that the user experience for applications is not affected in any way, except when TrustBase prevents an application from establishing an insecure connection.

### 2.4.3 Architecture

The architecture for TrustBase is given in Figure 2.1. These components are described below:

#### 2.4.3.1 Traffic Interceptor

The traffic interceptor intercepts all network traffic and delivers it to registered handlers for further processing. The interceptor is generic, lightweight, and can provide traffic to any

type of handler. Traffic for any specific stream is intercepted only as long as a handler is interested in it. Otherwise, traffic is routed normally.

The traffic interceptor is needed to secure existing applications. If a developer is willing to modify her application to call TrustBase directly for certificate validation, then she can use the validation API. Administrators can configure TrustBase to not intercept traffic from applications using this API.

#### 2.4.3.2 Handlers

Handlers are state machines that examine a traffic stream to isolate data used for authenticating connections and then pass this data to the policy engine. Data provided to the policy engine includes everything from the relevant protocol that is intercepted.[3] For example, with TLS this includes the `ClientHello` and `ServerHello` data in addition to the server certificate chain and the server hostname. The handler will allow or abort the connection, based on the policy engine's response.

TrustBase currently has both a TLS handler and an opportunistic TLS handler (e.g., STARTTLS), and due to the design of the traffic interceptor it is easy to add support for new secure transport protocols as they become popular (e.g., QUIC, DTLS).

#### 2.4.3.3 Policy Engine

The policy engine is responsible for using the registered authentication services to validate the server certificate extracted by the handler. The policy engine also aggregates validation responses if there are multiple active authentication plugins. The policy is configured by the system administrator, with sensible operating system defaults for ordinary users.

When the policy engine receives a validation request from a handler, it will query each of the registered authentication services to validate the server's certificate chain and host data. Authentication services can respond to this query in one of four ways: *valid*, *invalid*,

[3]This enables plugins to provide authentication methods that utilize TLS extensions and cipher suite information.

*abstain*, or *error*. Abstain and error responses are mapped to the valid or invalid responses, as defined in a configuration file.

To render a decision, the policy engine classifies plugins as either "necessary" or "voting", as defined in the configuration file. All plugins in the "necessary" category must indicate the certificate is valid, otherwise the policy engine will mark the certificate as invalid. If the necessary plugins validate a certificate, the responses from the remaining "voting" plugins are tallied. If the aggregation of valid votes is above a preconfigured threshold, the certificate is deemed valid by the policy engine. A write-protected configuration file lists the plugins to load, assigns each plugin to an aggregation group ("necessary" or "voting"), defines the timeout for plugins, etc.

#### 2.4.3.4 Plugin API

TrustBase defines a robust plugin API that allows a variety of authentication services to be used with TrustBase. The policy engine queries each authentication service by supplying host data and a certificate chain, and the authentication service returns a response. We provide both an asynchronous plugin API and a synchronous plugin API to facilitate the needs of different designs.

The synchronous plugin API is intended for use by simple authentication methodologies. Plugins using this API may optionally implement `initialize` and `finalize` functions for any setup and cleanup they need to perform. For example, a plugin may want to store a cache or socket descriptor for long-term use during runtime. Each plugin must also implement a `query` function, which is passed a data object containing a query ID, hostname, IP address, port, certificate chain, and other relevant context. The certificate chain is provided to the plugin DER encoded and in openssl's `STACK_OF(X509)` format for convenience. The `query` function returns the result of the plugin's validation of the query data (valid, invalid, abstain, or error) back to the policy engine.

The asynchronous plugin API allows for easier integration with more advanced designs, such as multithreaded and event-driven architectures. This API supplies a callback function through the `initialize` function that plugins must use to report validation decisions, using the query ID supplied by the data supplied to `query`. Thus the `initialize` function is required so that plugins may obtain the callback pointer (the `finalize` function is still optional). Asynchronous plugins also implement the `query` function, but return a status code from this function immediately and instead report their validation decision using the supplied callback.

#### 2.4.3.5 Validation API

The validation API provides a direct interface to the policy engine for certificate validation. New or modified applications can use this API to simplify validation, avoid common developer mistakes, and take advantage of TrustBase authentication services. Applications can use the API to validate certificates or request pinning for a self-signed certificate. The API also allows the application to receive validation error messages from TrustBase, allowing it to display errors directly in the application (TrustBase displays notifications through the operating system).

### 2.4.4 Addressing Certificate Pinning

Some applications have implemented certificate pinning to provide greater security in cases where the hosts that the application visits are static and known, rather than using the CA system for certificate validation. TrustBase wants to avoid the situation where its authentication services declare a certificate to be invalid when the application has validated it with pinning, but should also adhere to its core tenant that the system administrator should have ultimate control over how certificates are validated. Our measurements indicate that this circumstance is rare and affects relatively few applications, since the problem only arises when a certificate offered by a host does not also validate by the CA system (e.g., a self-signed

certificate). In the short term, TrustBase solves this problem by using the configuration file to allow whitelisting of programs that should bypass TrustBase's default policies. In the long term, this problem is solved by applications migrating to the validation API.

### 2.4.5 TLS 1.3 and Overriding the CA System

There are two situations where TrustBase cannot use default traffic interception to accomplish its primary goals. First, when an application uses TLS 1.3, the certificates that are exchanged are encrypted, preventing TrustBase from using passive traffic interception to independently validate certificates. Second, in some cases a system administrator may want to distrust the CA system entirely and rely solely on alternative authentication services. For example, the administrator may want to force applications currently using CA validation to accept self-signed certificates that have been validated using a notary system such as Convergence[55], or she may want to use DANE[39] with trust anchors that differ from those shipped with the system. When this occurs, TrustBase will use the new authentication service and determine the certificate is valid and allow a connection as configured by the administrator, but applications using the CA system may reject the certificate and terminate the connection. We stress that such a policy would not be intended to override strong certificate checks done by a browser (e.g., when communicating with a bank), but to provide a path for migrating away from the CA system as stronger alternatives emerge.

To handle both TLS 1.3 and overriding the CA system, TrustBase provides two options. The preferred option is to modify applications to rely on TrustBase for certificate validation, rather than performing their own checks. This is facilitated by the validation API described above. This enables new or modified applications to use the full set of authentication services provided by TrustBase in a natural manner.

A second option is to employ a local TLS proxy that can coerce existing applications that rely on the CA system to use new authentication services instead. The use of a proxy also allows TrustBase plaintext access to the server's certificate under TLS 1.3. TrustBase

gives the administrator the option of running such a proxy, but it is activated only in those cases where it is needed, namely when the policy engine determines a certificate is valid but the CA system would reject it. The proxy employed is a modified fork of `sslsplit` [78] and has shown itself to be scalable and performant in our experimentation. Note that in most cases this is not needed—for example, under Convergence, the certificates validated by notaries would likely also be validated by the CA system unless the certificate was self-signed, which is a situation likely to exist until CA alternatives gain significant traction. Given the vulnerabilities noted recently with proxies [24] administrators should exercise caution using this feature. Due to the features of the Windows root store, TrustBase on Windows can override the CA system without the use of a local proxy, as explained in Section 2.7.4.

### 2.4.6 Operating System Support

We designed the TrustBase architecture so that it could be implemented on additional operating systems. The main component that may need to be customized for each operating system is the traffic interception module. We are optimistic that this is possible because the TCP/IP networking stack and sockets are used by most operating systems.

Our Linux implementation is described in the following section. We also have a working prototype of TrustBase for Windows, which uses the Windows Filtering Platform API.

Mac OSX provides a native interface for traffic interception between the TCP and socket levels of the operating system. Apple's Network Kernel Extensions suite provides a "Socket Filter" API that could be used as the traffic interceptor.

For iOS, Apple provides a Network Extension framework that includes a variety of APIs for different kinds of traffic interception. The App Proxy Provider API allows developers to create a custom transparent proxy that captures application network data. Also available is the Filter Data Provider API that allows examination of network data with built-in "pass/block" functionality.

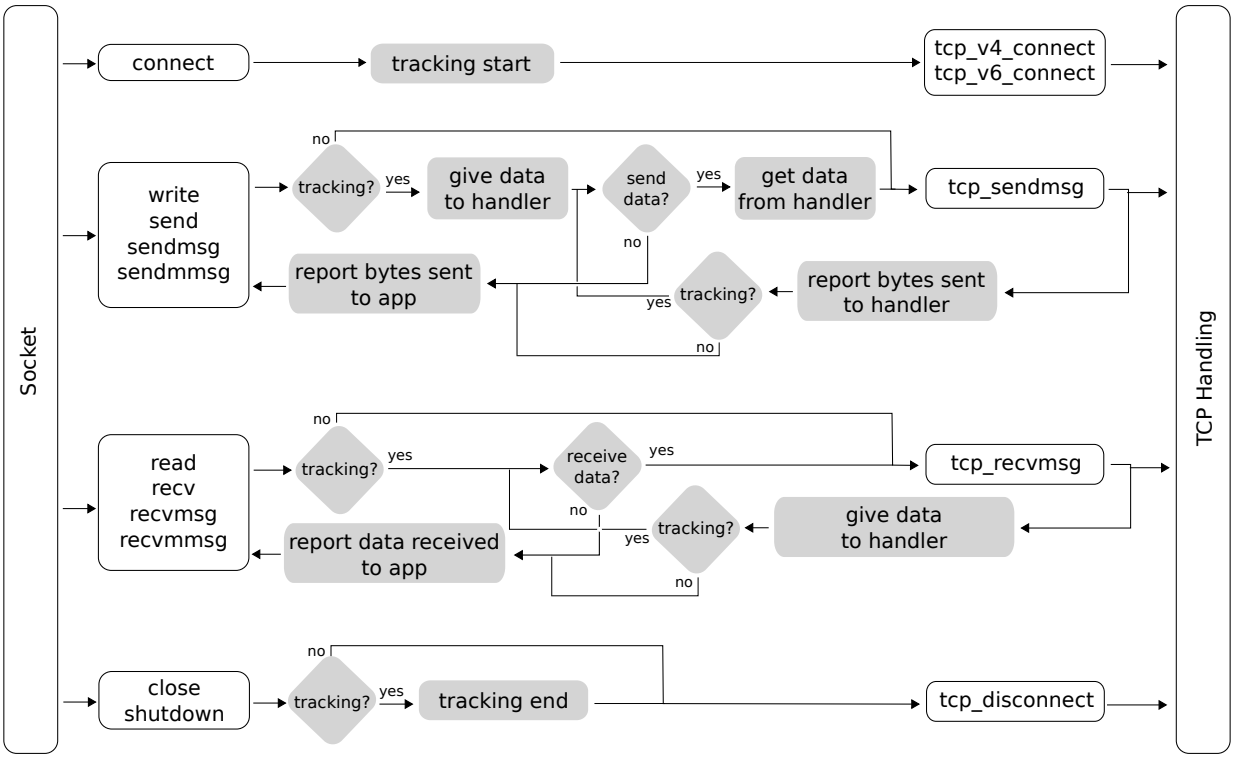



Figure 2.2: Linux Traffic Interceptor simplified flowchart. Grey boxes correspond to hooks for handlers, white boxes are native system calls and kernel functions

Because Android uses a variant of the Linux kernel, we believe our Linux implementation could be ported to Android with relative ease. We have a prototype of TrustBase on Android that instead uses the `VPNService` to intercept traffic.

## 2.5 Linux Implementation

We have designed and built a research prototype of TrustBase for Linux. The source code is available at `owntrust.org`.

We have developed a loadable kernel module (LKM) to intercept traffic at the socket layer, as data transits between the application and TCP handling kernel code. No modification of native kernel code is required, and the LKM can be loaded and unloaded at runtime. Similarly to how Netfilter operates at the IP layer, TrustBase can intercept traffic at the socket layer, before data is delivered for TCP handling, and pass it to application-level

programs, where it can be (optionally) modified and then passed back to the native kernel code for delivery to the original TCP functionality. Likewise, interception can occur after data finishes TCP processing and before it is delivered to the application. This enables TrustBase to efficiently intercept TLS connections in the operating system and validate certificates in the application layer.

The following discussion highlights the salient features of our implementation.

### 2.5.1 Traffic Interceptor

TrustBase provides generic traffic interception by capturing traffic between sockets and the TCP protocol. This is done by hooking several kernel functions and wrapping them to add traffic interception as needed. An overview of which functions are hooked and how they are modified is given in Figure 2.2. Items in white boxes on the left side of the figure are system calls. Items in white boxes on the right side of the figure are the wrapped kernel functions. The additional logic added to the native flow of the kernel is shown by the arrows and gray boxes in Figure 2.2.

When the TrustBase LKM is loaded, it hooks into the native TCP kernel functions whose pointers are stored in the global kernel structures `tcp_prot` (for IPv4) and `tcpv6_prot` (for IPv6). When a user program invokes a system call to create a socket, the function pointers within the corresponding protocol structure are copied into the newly-created kernel socket structure, allowing different protocols (TCP, UDP, TCP over IPv6, etc.) to be invoked by the same common socket API. The function pointers in the protocol structures correspond to basic socket operations such as sending and receiving data, and creating and closing connections. Application calls to `read`, `write`, `sendmsg`, and other system calls on that socket then use those protocol functions to carry out their operations within the kernel. Note that within the kernel, all socket-reading system calls (`read`, `recv`, `recvmsg`, and `recvmmsg`) eventually call the `recvmsg` function provided by the protocol structure. The same is true for the corresponding socket write system calls, as each result in calling the kernel `sendmsg`

function. When the LKM is unloaded, the original TCP functionality is restored in a safe manner.

From top to bottom in Figure 2.2, the functionality of the traffic interceptor is as follows. First, a call to `connect` informs the handler that a new connection has been created, and the handler can choose to intercept its traffic.

Second, when an application makes a system call to send data on the socket, the interceptor checks with the handler to determine if it is tracking that connection. If so, it forwards the data to the handler for analysis, and the handler chooses what data (potentially modified by the handler), if any, to relay to native TCP-sending code. After attempting to send data, the interceptor informs the handler how much of that data was successfully placed into the kernel's send buffer and provides notification of any errors that occurred. At this point the interceptor allows the handler to send additional data, if desired. This process continues until the handler indicates it no longer wishes to send data. The interceptor then queries the handler for the return values it wishes to report to the application (such as how many bytes were successfully sent or an error value) and these values are returned to the application.

Third, a similar, reversed process is followed for the reception of data from the network. If the interceptor is tracking the connection it can choose whether to receive data processed by TCP handling. Any data received is reported to the handler, which can choose whether to report a different value to the application. Note that handlers are allowed to report arbitrary values to applications for the amount of data sent or received, including false values, to allow greater flexibility in connection handling, or to maintain application integrity when injecting additional bytes into a stream. For example, to provide more time to obtain and parse a message, a handler may indicate to an application that zero bytes have been received on a nonblocking socket, even though some or all of the data may have already been received. After the handler has completed its operation it can report to a subsequent receive call from the application that bytes were received, and fill the application's provided buffer with relevant

data. As another example, if a handler wishes to append data to a message successfully transferred to the OS by an application using the `send` system call, it should enforce that the return value of this function be the number of bytes the application expects to have been sent, rather than a higher number that includes the added bytes.

Finally, a call to `close` (on the last remaining socket descriptor for a connection) or `shutdown` informs the handler that the connection is closed. Note that the handler may also choose to abandon tracking of connections before this point.

Handlers for various network observation and modification can be constructed by implementing a small number of functions, which will be invoked by the traffic interceptor at runtime. These functions roughly correspond to the grey boxes in Figure 2.2. For example, handlers must implement functions to send and receive data, indicate whether to continue or cease tracking of a connection, etc. The traffic interceptor calls these functions to provide the handler with data, receive data from the handler to be forwarded to applications or remote hosts, and other tasks. Such an architecture allows developers to implement arbitrary protocol handlers as simple finite state machines, as demonstrated by the TLS handler and opportunistic TLS handlers described in the following subsections.

Another option for implementing traffic interception would have been to use the Netfilter framework, but this is not an optimal approach. TrustBase relies on parsing traffic at the application layer, but Netfilter intercepts traffic at the IP layer. For TrustBase to be implemented using Netfilter, TrustBase would need to transform IP packets into application payloads. This could be done either by implementing significant portions of TCP, including out-of-order handling and associated buffers, or passing traffic through the network stack twice, once to parse the IP packets for TrustBase and once for forwarding the traffic to the application. Both of these options are problematic, creating development and performance overhead, respectively.

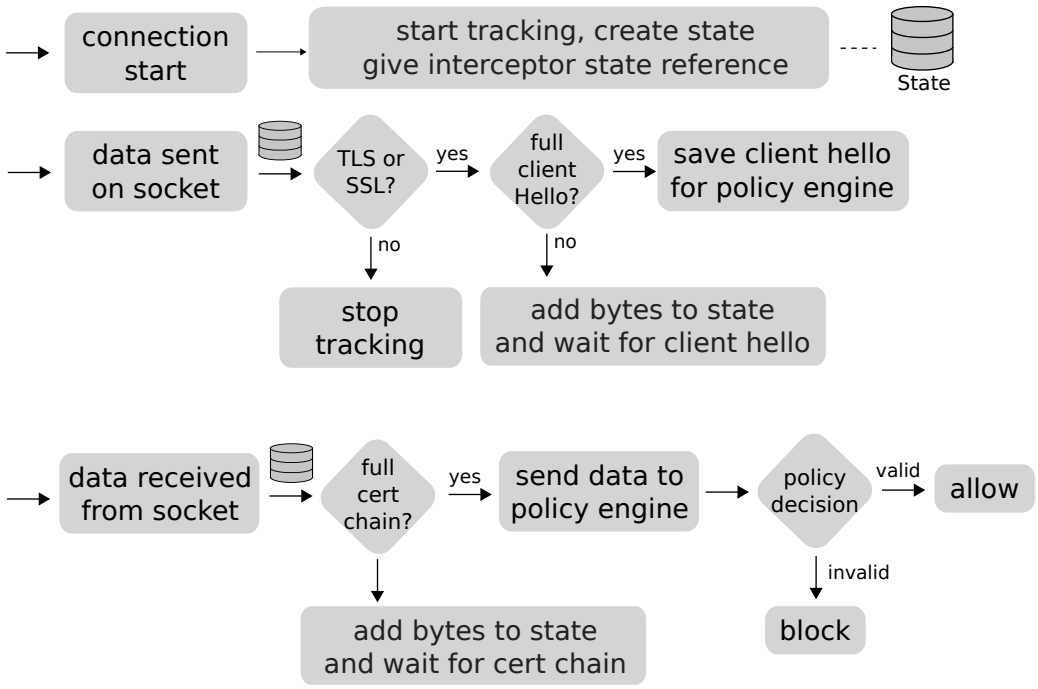



Figure 2.3: Simplified view of TLS handler

### 2.5.2 TLS Handler

TrustBase includes a handler for the traffic interceptor dubbed the "TLS handler". The TLS handler extracts certificates from TLS network flows and forwards them to the policy engine for validation.

Figure 2.3 provides a high-level overview of how this handler operates. When a new socket is created, the handler creates state to track the connection, which the handler will have access to for all subsequent interactions with the interceptor. The destination IP address and port of the connection and PID of the application owning the connection are provided to the handler during connection establishment by the interceptor. Since the handler is implemented in a LKM, the PID of the socket can be used to obtain any further information about the application such as the command used to run it, its location, and even memory contents.

When data is sent on the socket, the handler checks state data to determine whether the connection has initiated a TLS handshake. If so, then it expects to receive a `ClientHello`; the handler saves this message for the policy engine so that it can obtain the hostname of the desired remote host, if the message contains a Server Name Indication (SNI) extension. If SNI is not used, a log of applications' DNS lookups can be used to infer the intended host,[4] similar to work by Bates et al. [10]

When data is received on the socket, the TLS handler waits until it has received the full certificate chain, then it sends this chain and other data to the policy engine for parsing and validation.

Note, the TLS handler understands the TLS record and handshake protocols but does not perform interpretations of contained data. This minimizes additions to kernel-level code and allows ASN.1 and other parsing to be done in userspace by the policy engine.

### 2.5.3 Opportunistic TLS Handler

We have also implemented an opportunistic TLS handler, which provides TrustBase support for plaintext protocols that may choose to upgrade to TLS opportunistically, such as STARTTLS. This handler performs passive monitoring of plaintext protocols (e.g., SMTP), allowing network data to be fast-tracked to and from the application and does not store or aggressively process any transiting data. If at some point the application requests to initiate a TLS connection with the server (e.g., via a STARTTLS message), the handler processes acknowledgments from the server and then delivers control of the connection to the normal TLS handler, which is free to handle the connection as if it were conducting regular TLS.

It should be noted that the use of opportunistic TLS protocols by applications is subject to active attackers who perform stripping attacks to make the client believe the server does not support TLS upgrades, an existing vulnerability well documented by recent work [23, 33, 42]. TrustBase can prevent this type of attack, as discussed in Section 2.6.

[4]Our experimentation showed that all popular TLS implementations and libraries now use SNI, and Akamai reports HTTPS SNI global usage at over 98% [61], so this fallback mechanism is almost never needed.

### 2.5.4 Policy Engine

The policy engine receives raw certificate data from the TLS handler and then validates the certificates using the configured authentication services. To avoid vulnerabilities that may arise from performing parsing and modification of certificates in the kernel, all such operations are carried out in user space by the policy engine.

Communication between TrustBase kernel space and user space components is conducted via Netlink, a robust and efficient method of transferring data between kernel and user space, provided natively by the Linux kernel. The policy engine asynchronously handles requests from the kernel module, freeing up the kernel threads to handle other connections while a response is constructed.

Native plugins must be written in either C or C++ and compiled as a shared object for use by the policy engine. However, in addition to the plugin API, TrustBase supports an addon API that allows plugins to be written in additional languages. Addons provide the code needed to interface between the native C of the policy engine and the target language it supports. We have implemented an addon to support the Python language and have created several Python plugins.

## 2.6 Security Analysis

The TrustBase architecture, prototype implementation, and sample plugins have many implications for system security. In this section we provide a security analysis of the centralized system design, application coverage, protection of applications from attackers, and protection of TrustBase itself from attackers.

### 2.6.1 Centralization

Concentrating certificate validation in an operating system service has some risks and benefits. Any vulnerability in the service has the potential to impact all applications on the system. An exploit that grants an attacker root permission leads to compromise of the host. An exploit

that causes a certificate to be rejected when it should be accepted is a type of denial-of-service attack. We note that if an attacker is able to get TrustBase to accept a certificate when it should not, any application that does its own certificate authentication correctly will be unaffected. If the application is broken, the TrustBase failure will not make the situation any worse than it already was. The net effect is a lost opportunity to make it better.

The risks of centralization are common to any operating system service. However, centralization also has a compelling upside. For instance, all of our collective effort can be centered on making the design and implementation correct, and all applications can benefit.[5] Securing a single service is more scalable than requiring developers to secure each application or library independently. It also enforces an administrator's preferences regardless of application behavior. Additionally, when a protocol flaw is discovered, it can be more rapidly tested and patched, compared to having to patch a large number of applications.

### 2.6.2 Coverage

Since one of the goals of TrustBase is to enforce proper certificate validation on all applications on a system, the traffic interceptor is designed to stand between the transport and application layers of the OS so that it can intercept and access all TLS flows from local applications. The handlers associated with the traffic interception component are made aware of a connection when a `connect` call is issued and can associate that connection with all data flowing through it. Applications that utilize their own custom TCP/IP stack must utilize raw sockets, which require administrator privileges and are therefore implicitly trusted by TrustBase.

To obtain complete coverage of TLS, our handlers need only monitor initial TLS handshakes (standard TLS) and the brief data preceding them (STARTTLS). The characteristics of TLS renegotiation and session termination are compatible with our approach.

In TLS renegotiation, subsequent handshakes use key material derived using the master secret of the first handshake. Thus if the policy engine correctly authenticates and

[5] All applications would likewise benefit from caching among authentication services.

validates the first handshake, TLS renegotiations are implicitly verified as well. Attackers who obtained sufficient secrets to trigger a renegotiation, through some other attack on the TLS protocol or implementation (outside our threat model), have no need to take advantage of renegotiation as they have complete control over the connection already. We also note that renegotiation is rare and typically used for client authentication for an already authenticated server, and has become less relevant for SGC or refreshing keys [77].

Session termination policies for TLS allow us to associate each TLS session with only one TCP connection. In TLS, a close notify must be immediately succeeded by a responding close notification and a close down of the connection [20]. Subsequent reconnects to the target host for additional TLS communication are detected by the TrustBase traffic interceptor and presented to the handlers. We have found that TLS libraries and applications do indeed terminate a TCP session when ending a TLS session, although many of them fail to send an explicit TLS close notification and rely solely on TCP termination to notify the remote host of the session termination.

### 2.6.3 Threat Analysis

The coverage of TrustBase enables it to enforce both proper and additional certificate validation procedures on TLS-using applications. There are a variety of ways that attackers may try perform a TLS MITM against these applications. A selection of these methods and discussion of how TrustBase can protect against them follows. For each, we verified our solution utilizing an "attacker" machine acting as a MITM using sslsplit [78], and a target "victim" machine running TrustBase. For some scenarios, the victim machine was implanted with our own CA in the distribution's shipped trust store or the store of a local user or application. Applications tested utilize the tools and libraries mentioned in section 2.7.2.

- **Hacked or coerced certificate authorities:** Attackers who have received a valid certificate through coercion, deception, or compromise of CAs are able to subvert even proper CA validation. Under TrustBase, administrators can choose to deploy pinning

or notary plugins, which can detect the mismatch between the original and forged certificate, preventing the attacker from initiating a connection. We have developed plugins that perform these actions and verified that they prevent such attacks.

- **Local malicious root:** Attackers utilizing certificates that have been installed into an application or user trusted store will be trusted by many target applications. Even Google Chrome will ignore certificate pins in the presence of a certificate that links back to a locally-installed root certificate. TrustBase can protect against this by utilizing similar plugins to the preceding scenario.
- **Absence of status checking:** Many applications still do not check OCSP or Certificate Revocation Lists to determine if a received certificate is valid [51]. In these cases, attackers utilizing stolen certificates that have been reported can still perform a MITM. Administrators who want to prevent this from happening can add an OCSP or CRL plugin to the policy engine and ensure these checks for all applications on the machine. We have developed both OCSP and CRLSet plugins and verified that they perform status checks where applicable. For example, the OSCP plugin can be used to check certificates received by the Chrome browser, which does not do this natively.
- **Failure to validate hostnames:** Some applications properly validate signatures from a certificate back to a trusted root but do not verify that the hostname matches the one contained in the leaf certificate. This allows attackers to utilize any valid certificate, including those for hosts they legitimately control, to intercept traffic [35]. The TrustBase policy engine strictly validates the common name and all alternate names in a valid certificate against the intended hostname of the target host to eliminate this problem.
- **Lack of validation:** For applications that blindly accept all certificates, attackers need only send a self-signed certificate they generate on their own, or any other for which they have the private key, to MITM a connection. TrustBase prohibits this by

default, as the policy engine ensures the certificate has a proper chain of signatures back to a trust anchor on the machine and performs the hostname validation described previously.

- **STARTTLS downgrade attack:** Opportunistic TLS begins with a plaintext connection. A downgrade attack occurs when an active attacker suppresses STARTTLS-related messages, tricking the endpoints into thinking one or the other does not support STARTTLS. The net result is a continuation of the plaintext connection and possible sending of sensitive data (e.g., email) in the clear. TrustBase mitigates this attack by an option to enforce STARTTLS use. When STARTTLS is used to communicate with a given service, TrustBase records the host information. Future connections to that host are then required to upgrade via STARTTLS. If the host omits STARTTLS and prohibits its use, the connection is severed by TrustBase to prevent leaking sensitive information to a potential attacker.[6] TrustBase also allows the system administrator to configure a strict TLS policy, which disallows plaintext connections even if it has no prior data about whether a remote host supports STARTTLS.

### 2.6.4 Hardening

The following design principles strengthen the security of a TrustBase implementation. First, the traffic interceptor and handler components run in kernel space. Their small code size and limited functionality—handlers are simple finite state machines—make it more likely that formal methods and source code auditing will provide greater assurance that an implementation is correct. Second, the policy engine and plugins run in user space. This is where error-prone tasks such as certificate parsing and validation occur. The use of privilege separation [74] and sandboxing [53] techniques can limit the potential harm when any of these components is compromised. Third, plugins can only be installed and configured by an

[6]This could be further strengthened by checking DANE records to determine if the server supports STARTTLS. We are likewise interested in pursuing whether this technique can be used to protect against other types of downgrade attacks.

administrator, which prohibits unprivileged adversaries and malware from installing malicious authentication services. Finally, communications between the handlers, policy engine, and plugins are authenticated to prevent local malware from spoofing a certificate validation result.

TrustBase is designed to prevent a local, nonprivileged user from inadvertently or intentionally compromising the system. (1) Only privileged users can insert and remove the TrustBase kernel module, prohibiting an attacker from simply removing the module to bypass it. The same is true for plugins. (2) The communication between the kernel module component of TrustBase and the user space policy engine is performed via a custom Generic Netlink protocol that protects against nonprivileged users sending messages to the kernel. The protocol definition takes advantage of the Generic Netlink flag `GENL_ADMIN_PERM`, which enforces that selected operations associated with the custom protocol can only be invoked by processes that have administrative privileges for networking (the capability mapped to `CAP_NET_ADMIN` in Linux systems). This prevents a local attacker from using a local Netlink-utilizing process to masquerade as the policy engine to the kernel. (3) The policy engine runs as a nonroot, `CAP_NET_ADMIN` process that can be invoked only by a privileged user. (4) The configuration files, plugin directories, and binaries for TrustBase are write-protected to prevent unauthorized modifications from nonprivileged users. This protects against weakening of the configuration, disabling of plugins, shutting down or replacing the policy engine, or enabling of bogus plugins.

TrustBase stops traffic interception for a given flow as soon as it is identified as a non-TLS connection. Experimental results show that TrustBase has negligible overhead with respect to memory and time while tracking connections. Thus it is unlikely that an attacker could perform a denial-of-service attack on the machine by creating multiple network connections, TLS or otherwise, any easier than in the non-TrustBase scenario. Such an attack is more closely associated with firewall policies.

An attacker may seek to compromise TrustBase by crafting an artificial TLS handshake that results in some type of TrustBase failure, hoping to cause some kind of application error or termination. We reduce this attack surface by performing no parsing in the kernel except for TLS handshake records, which involves just the message type, length, and version headers. ASN.1 and other data sent to the policy engine are evaluated and parsed by standard openssl functions, which have undergone widespread scrutiny and use for many years. The TrustBase code has been made publicly available, and we invite others to audit the code. We note that, in the absence of the local proxy, TrustBase will not coerce an application to accept a certificate that the application would normally reject.

## 2.7 Evaluation

We evaluated the prototype of TrustBase to measure its performance, ensure compatibility with applications, and test its utility for deploying authentication services that can harden certificate validation.

### 2.7.1 Performance

To measure the overhead incurred by TrustBase, we instrumented our implementation to record the time required to establish a TCP connection, establish a TLS connection, and transfer a file of varying size (2MB - 500 GB). We tested TrustBase with two plugins, CA Validation and Certificate Pinning (see Section 2.7.5). The target host for these connections was a computer on the same local network as the client machine, to reduce the effect of latency and network noise. The host presented a valid certificate chain that also employed an intermediate authority, representing a realistic circumstance for web browsing and forcing plugins to execute all of their validity checks. Our testing used a modern PC running Fedora 21 and averaged across 1,000 trials.

Figure 2.4 shows boxplots that characterize the timing of TCP and TLS handshakes, with and without TrustBase active. There is no discernible difference for TCP handshake

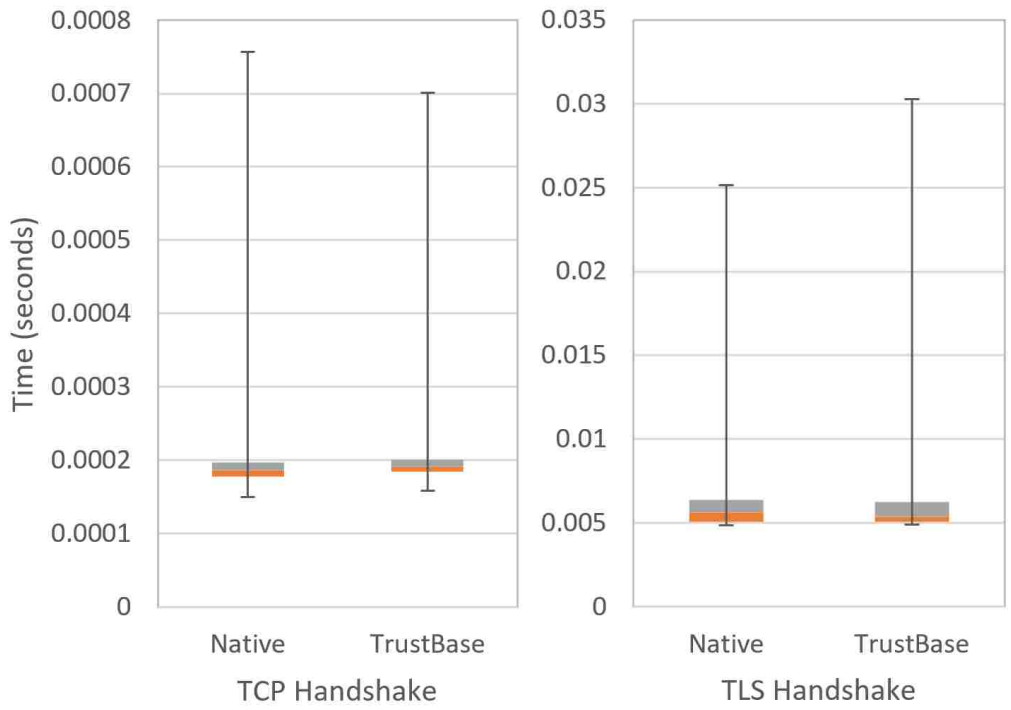



Figure 2.4: Handshake Timings for TCP (left) and TLS (right) handshakes with and without TrustBase running.

timings and the average difference is less than 10 microseconds, with neither configuration consistently beating the other in subsequent experiments. This is expected behavior because the traffic interceptor is extremely light-weight for TCP connections. Average TLS handshake times with and without TrustBase also have no discernible difference, with average handshake times for this experiment of 5.9 ms and 6.0 ms, respectively. Successive experiments showed again that neither average consistently beat the other. This means that the inherent fluctuations in system and network conditions account for more time than the additional control paths TrustBase introduces. This is also expected, as the brevity of TLS handling code, its place in the kernel, the use of efficient Netlink transport and other design choices were made with performance in mind.

Our experimentation with varying file sizes also exhibited no difference between native and TrustBase cases. Note that the TrustBase timings for the TLS handshake may increase if a particular plugin is installed that requires more processing time or relies on Internet queries to function, and that this overhead is inherent to that service and not the TrustBase core.

The memory footprint in our Linux prototype is also negligible. For each network connection, TrustBase temporarily stores less than 300 bytes of data, plus the length of any TLS handshake messages encountered. Connections not using TLS use even less memory than this and carry a zero-byte memory overhead once their nature has been determined and TrustBase ceases to monitor them.

### 2.7.2 Compatibility

One goal of TrustBase is to strengthen certificate authentication for existing, unmodified applications and to provide additional authentication services that strengthen the CA system. To meet this goal, TrustBase must be able to enforce proper authentication behavior by applications, as defined by the system administrator's configuration.

There are three possible cases for the policy engine to consider. (1) If a certificate has been deemed valid by both TrustBase and the application, the policy engine allows the original certificate data to be forwarded on to the application, where it is accepted naturally. (2) In the case where the application wishes to block a connection, regardless of the decision by TrustBase, the policy engine allows this to occur, since the application may have a valid reason to do so. We discuss in Section 2.4.5, the special case when a new authentication service is deployed that wishes to accept a certificate that the CA system normally would not. (3) In the case where validation with TrustBase fails, but the application would have allowed the connection to proceed, the policy engine blocks the connection by forwarding an intentionally invalid certificate to the application, which triggers any SSL application validation errors an application supports, and then subsequently terminates the connection.

We tested TrustBase with 34 popular applications and libraries and tools, shown in Table 2.1.[7] TrustBase successfully intercepted and validated certificates for all of them. For each library tested, and where applicable, we created sample applications that performed no validation and improper validation (bad checking of signatures, hostnames, and validity

[7]These are a superset of the tools and libraries tested with CertShim

| Library | Tool |
|---|---|
| C++ | gnutls-cli |
| libcurl | curl |
| libgnutls | sslscan |
| libssl | openssl s_client |
| libnss | openssl s_time |
| Java | lynx |
| SSLSocketFactory | fetchmail |
| Perl | firefox |
| socket::ssl | chrome/chromium |
| PHP | mpop |
| fsockopen | w3m |
| php_curl | ncat |
| Python | wget |
| httplib | steam |
| httplib2 | thunderbird |
| pycurl | kmail |
| pyOpenSSL | pidgin |
| python ssl | |
| urllib, urllib2, urllib3 | |
| requests | |

Table 2.1: Common Linux libraries and tools compatible with TrustBase

dates). We then verified that TrustBase correctly forced these applications to reject false certificates despite those vulnerabilities in each case. In addition, we observed that TrustBase caused no adverse behavior, such as timeouts, crashes, or unexpected errors.

### 2.7.3 Android Prototype

To verify that the TrustBase approach works on mobile platforms and is compatible with mobile applications, we built a prototype for Android. Source code can be found at `owntrust.org`.

Our Android implementation uses the `VPNService` so that it can be installed on an unaltered OS and without root permissions. The drawback of this choice is that only one VPN service can be active on the Android at a time. In the long-term, adding socket-level interception to the Android kernel would be the right architectural choice, and then TrustBase could use similar traffic interception techniques as with the Linux implementation.

| | |
|---|---|
| **CA Validation** | Enforces standard certificate validation using `openssl` functions and standard practices for validating hostnames, Basic Constraints, dates, etc. |
| **Whitelist** | Stores a set of certificates that are always considered valid for their respective hosts, such as self-signed certificates. |
| **Certificate Pinning** | Uses Trust On First Use to pin certificates for any host; expired certificates are replaced by the next certificate received by a connection to that domain. |
| **Certificate Revocation** | Checks OCSP to determine whether the certificate has been revoked. |
| **CRLSet Blocking** | Checks Google's CRLSet to determine whether the certificate has been blocked, extending Chrome's protection to all apps. |
| **DANE** | Uses the DNS system to distribute public keys in a TLSA record [39]. |
| **Notary** | Based on ideas presented by Perspectives [89] and Convergence [55], it connects securely to one or more notary servers to validate the certificate received by the client is the same one that is seen by the notaries. |
| **Cipher Suite Auditor** | Uses Client Hello and Server Hello information, along with a configuration with secure defaults, to disallow weak cipher suites. It can also require that certain TLS extensions be employed (e.g., TACK[56]). |

Table 2.2: Authentication and Security services implemented with TrustBase

The primary engineering consequence of using the `VPNService` on Android is that TrustBase must intercept IP packets from applications but emit TCP (or UDP) packets to the network. If it could use raw sockets, then TrustBase could merely transfer IP packets between the `VPNService` and the remote server. Unfortunately, the lowest level socket endpoint an Android developer can create is the Java `Socket` or `DatagramSocket`, which encapsulate TCP and UDP payloads respectively. Therefore, we must emulate IP, UDP and TCP to facilitate communication between the `VPNService` and the sockets used to communicate with remote hosts. For TCP, this involves maintaining connection state, emulating reliability, and setting appropriate flags (SYN, ACK, etc.) for TCP traffic.

To verify compatibility with mobile applications, we tested 16 of the most popular Android applications: Chrome, YouTube, Pandora, Gmail, Pinterest, Instagram, Facebook, Google Play Store, Twitter, Snapchat, Amazon Shopping, Kik, Netflix, Google Photos, Opera, and Dolphin. TrustBase on Android successfully intercepted and strengthened certificate validation for all of them.

### 2.7.4 Windows Prototype

To demonstrate that the TrustBase approach works on Windows, we also built a prototype for Windows 10. Source code can be found at `owntrust.org`.

The traffic interceptor component of TrustBase on Windows is implemented utilizing the native Windows Filtering Platform (WFP) API, acting as a kernel-mode driver. Reliance on the WFP reduced the code necessary to provide traffic interception capabilities and also made them easy to maintain, given that the Windows kernel code is not open source. As on Linux, this kernel code is event-driven, collects connection information, and transmits it to a userspace policy engine for processing and decision making. The policy engine is patterned after its Linux counterpart, supports both Python and C plugins, and uses native Windows libraries where possible (e.g., Microsoft's CryptoAPI and native threading APIs).

The nature of the Windows root certificate store allows TrustBase to avoid utilizing a TLS Proxy in cases where overriding the CA system is desired (see Section 2.4.5). Windows has the ability to dynamically alter the root certificate store during runtime, and applications using the CA system will be immediately subject to those changes. This allows TrustBase to dynamically add self-signed certificates to the root store when the policy engine deems them trustworthy. Through this mechanism TrustBase can override the CA system by placing a validated certificate in the root store before the application obtains and validates it against the root store. TrustBase maintains identifying hashes of all the certificates added to the root store and removes them when the connections using them are terminated. As on Linux,

applications that use their own private certificate stores cannot have their validation rejections overridden using this methodology.

### 2.7.5 Utility

To validate the utility of TrustBase, we implemented eight useful authentication services. Table 2.2 describes each of these services. These services illustrate the types of control that TrustBase can provide to an administrator in securing TLS on a system. The CA validation plugin ensures that all applications on the system perform appropriate checks when validating certificates received through TLS (hostname, basic constraints, expiration, etc.). The whitelist represents a more manual, customized approach to validation, likely to be used in conjunction with other services to handle edge cases. Our certificate pinning and certificate revocation services enforce more advanced checks that are usually reserved for individual applications but can now be deployed system-wide. Note that this includes the deployment of Google's CRLSets checks, which are normally reserved for Chromium browsers only. This addresses the limitation noted by [24] concerning the isolation of newer validation technologies in browser code. The Notary and DANE services can be leveraged to trust additional channels of information aside from CA signatures and revocation lists. Finally, our cipher suite auditor service allows system administrators to prevent connections that attempt to utilize weak cipher suites and signing algorithms, using the additional handshake information provided to all plugins.

## 2.8 Future Work

TrustBase explores the benefits and drawbacks of providing authentication as an operating system service and giving administrators control over how all authentication decisions on their machines are made. In doing so, a step has been taken toward empowering administrators to control secure connections on their machines. However, some drawbacks have been noted, such as the reliance on a local proxy to support TLS 1.3 interception and CA overriding in

some cases on Linux. These issues are caused by applications dictating the security of the machine's connections, using their own (or third party) security features and keys, reducing operating system and administrator control.

We are currently investigating further steps into this territory to provide great administrator control of security without some of these drawbacks. One such step is providing TLS as an operating system service, meaning that the operating system provides encryption for applications, not just authentication. Current TLS libraries are a burden on application developers, who are often not security experts. In addition, developers do not necessarily share the same security goals as the vendors or administrators who configure the systems upon which applications run. By providing TLS as an operating system service, application developers are relieved of this burden and the OS can invoke the TrustBase validation API natively. This removes the need for developers to explicitly invoke the validation API, and provides the OS with visibility and control over all TLS data, including TLS 1.3 handshakes, as the OS becomes the de facto TLS client. Such a measure enables system-wide deployment of security measures, such as cipher suite customization, TLS extension deployment, and responses to CVEs. This also allows OS vendors and system administrators an easier upgrade path for TLS versions.

Since network application developers are already familiar with the POSIX socket API, we are working on providing TLS as a protocol type in the socket API, the same way the OS provides TCP and UDP protocols as a service. In contrast to using a userspace library, this approach allows network application developers unfamiliar with security to operate in a well-known environment, utilizes an existing OS API that can be shared by many different platform implementations, and allows strict configuration and control by administrators. By creating a socket using a new `IPPROTO_TLS` parameter (as opposed to `IPPROTO_TCP`), developers can use the `bind`, `connect`, `send`, `recv`, and other socket API calls with which thy are already familiar, focusing solely on application data and letting the OS handle all TLS functionality. The generalized `setsockopt` and `getsockopt` are available to specify remote

hostnames and additional options to the OS TLS service without violating the existing socket API.

## 2.9 Conclusion

We have explored how to fix broken authentication in existing applications, while also providing a platform for improved authentication services. To solve these problems we used two guiding principles—centralizing authentication as an operating system service and giving system administrators control over authentication policy. Following these two principles, we designed the TrustBase architecture for certificate authentication, meeting our design goals of securing existing applications, strengthening the CA system, providing full application coverage, enabling universal deployment, and imposing negligible overhead. We have presented a research prototype for TrustBase on Linux, discussed how we hardened this implementation, provided a security analysis, and evaluated its performance. We have provided source code for Linux, Android, and Windows prototypes. Finally, we have written eight authentication services to demonstrate the utility of this approach, extending CA hardening to all applications.

# Chapter 3

# The Secure Socket API

In this chapter we explore placing TLS functionality in the operating system, simplifying the security API for application developers, and providing administrators full control of TLS configuration options. The text of this chapter is an extended version of the following published article:

> O'Neill, M., Heidbrink, S., Whitehead, J., Perdue, T., Dickinson, L., Collett, T., Bonner, N., Seamons, K., & Zappala, D. *The Secure Socket API: TLS as an Operating System Service*. In proceedings of USENIX Security Symposium, 2018.

## 3.1 Abstract

SSL/TLS libraries are notoriously hard for developers to use, leaving system administrators at the mercy of buggy and vulnerable applications. We explore the use of the standard POSIX socket API as a vehicle for a simplified TLS API, while also giving administrators the ability to control applications and tailor TLS configuration to their needs. We first assess OpenSSL and its uses in open source software, recommending how this functionality should be accommodated within the POSIX API. We then propose the Secure Socket API (SSA), a minimalist TLS API built using existing network functions and find that it can be employed by existing network applications by modifications requiring as little as one line of code. We next describe a prototype SSA implementation that leverages network system calls to provide privilege separation and support for other programming languages. We end with a discussion

of the benefits and limitations of the SSA and our accompanying implementation, noting avenues for future work.

### 3.2 Introduction

Transport Layer Security (TLS[1]) is the most popular security protocol used on the Internet. Proper use of TLS allows two network applications to establish a secure communication channel between them. However, improper use can result in vulnerabilities to various attacks. Unfortunately, popular security libraries, such as OpenSSL and GnuTLS, while feature-rich and widely-used, have long been plagued by programmer misuse. The complexity and design of these libraries can make them hard to use correctly for application developers and even security experts. For example, Georgiev et al. find that the "terrible design of [security library] APIs" is the root cause of authentication vulnerabilities [35].

Significant efforts to catalog developer mistakes and the complexities of modern security APIs have been published in recent years [14, 31, 38, 68, 85]. As a result, projects have emerged that reduce the size of security APIs [69], enhance library security [6], and perform certificate validation checks on behalf of vulnerable applications [10, 15, 32, 66]. A common conclusion of these works is that TLS libraries need to be redesigned to be simpler for developers to use securely.

In this work we present the Secure Socket API (SSA), a TLS API for applications designed to work within the confines of the existing standard POSIX socket API already familiar to network programmers. We extend the POSIX socket API in a natural way, providing backwards compatibility with the existing POSIX socket interface. This effort required an analysis of current security library use to guide our efforts, and careful interaction with kernel network code to not introduce undue performance overhead in our implementation. The SSA enables developers to quickly build TLS support into their applications and administrators to easily control how applications use TLS on their machines. We demonstrate

[1]Unless otherwise specified, we use TLS to indicate TLS and SSL

our prototype SSA implementation across a variety of use cases and also show how it can be trivially integrated into existing programming languages.

Our contributions are as follows:

- An analysis of contemporary use of TLS by 410 Linux packages and a qualitative breakdown of OpenSSL's 504 API endpoints for TLS functionality. These analyses are accompanied by design recommendations for the Secure Socket API, and may also serve as a guide for developers of security libraries to improve their own APIs.
- A description of the Secure Socket API and how it fits within the existing POSIX socket API, with descriptions of the relevant functions, constants, and administrator controls. We also provide example usages and experiences creating new TLS applications using the SSA that require less than ten lines of code and as little as one. We modify existing applications to use the SSA, resulting in the removal of thousands of lines of existing code.
- A description of and source code for a prototype implementation of the Secure Socket API. We also provide a discussion of benefits and features of this implementation, and demonstrate the ease of adding SSA support to other languages.
- A description of and source code for a tool that dynamically ports existing OpenSSL-using applications to use the SSA without requiring modification.

Previous findings have motivated the work for simpler TLS APIs and better administrator controls. This work explores utilization of the POSIX socket API as a possible avenue to address these needs.

We also discuss some finer points regarding the implementation and use of the SSA. We outline the benefits and drawbacks of our chosen implementation, and do the same for some suggested alternative implementations. For users of the SSA, we discuss the avenues for SSA configuration and its deployment with respect to different platforms and skill levels of users.

## 3.3 Motivation

TLS use by applications is mired by complicated APIs and developer mistakes, a problem that has been well documented. The `libssl` component of the OpenSSL 1.0 library alone exports 504 functions and macros for use by TLS-implementing applications. This problem is likely to persist, as the unreleased OpenSSL 1.1.1 has increased this number substantially. This and other TLS APIs have been criticized for their complexity [35, 38] and, anecdotally, our own explorations find many functions within `libssl` that have non-intuitive semantics, confusing names, or little-to-no use in applications. A body of work has cataloged developer mistakes when using these libraries to validate certificates, resulting in man-in-the-middle vulnerabilities [14, 31, 35].

A related problem is that the reliance on application developers to implement security inhibits the control administrators have over their own machines. For example, an administrator cannot currently dictate what version of TLS is used by applications she installs, what cipher suites and key sizes are used, or even whether applications use TLS at all. This coupling of application functionality with security policy can make otherwise desirable applications unadoptable by administrators with incompatible security requirements. This problem is exacerbated when security flaws are discovered in applications and administrators must wait for security patches from developers, which may not ever be provided due to project shutdown, financial incentive, or other reasons. Thus TLS connection security is at the mercy of application developers, despite their inability to properly use security APIs and unfamiliarity with the specific security needs of system administrators. One illustration of the demand for administrator control is the Redhat-led effort to create a system-wide "CryptoPolicy" configuration file [57]. Through custom changes in OpenSSL and GNUTLS, this configuration file allows developers to defer some security settings to administrators.

The synthesis of these two problem spaces is that developers lack a common, usable security API and administrators lack control over secure connections. In this paper we explore a solution space to this problem through the POSIX socket API and operating system

control. We seek to improve on prior endeavors by reducing the TLS API to a handful of functions that are already offered to and used by network programmers, effectively making the TLS API itself nearly transparent. This drastically reduces the code required to use TLS. We also explore supporting programming languages beyond C/C++ with a singular API implementation. Developers merely select TLS as if it were a built-in protocol such as TCP or UDP. Moreover, this enables administrators to configure TLS policies system-wide, while allowing developers to use options to add configuration and request stricter security policies.

Shifting control of TLS to the operating system and administrators may be seen as controversial. However, most operating systems already offer critical services to applications to reduce code redundancy and to ensure that the services are run in a manner that does not threaten system stability or security. For example, application developers on Linux and Windows are not expected to write their own TCP implementation for networking applications or to implement their own file system functionality when writing to a file. Moreover, operating systems and system administrators have been found to focus more attention on security matters [63]. Thus we believe establishing operating system and administrator control of TLS and related security policies is in line with precedent and best practice.

### 3.4 SSA Design Goals

Our primary goal in developing the SSA is to find a solution that is both easy to use for developers and grants a high degree of control to system administrators. Since C/C++ developers on Linux and other Unix-like systems already use the POSIX socket API to create applications that access the network, this API represents a compelling path for simplification of TLS APIs. Other languages use this API directly or indirectly, either through implementation of socket system calls or by wrapping another implementation. If TLS usage can be mapped to existing POSIX API syntax and semantics, then that mapping represents the most simple TLS API possible, in the sense that other approaches would either need to wrap or redefine the standard networking API.

Under the POSIX socket API, developers specify their desired protocol using the last two parameters of the `socket` function, which specify the type of protocol (e.g., `SOCK_DGRAM`, `SOCK_STREAM`), and optionally the protocol itself (e.g., `IPPROTO_TCP`), respectively. Corresponding network operations such as `connect`, `send`, and `recv` then use the selected protocol in a manner transparent to the developer. We explore the possibility of fitting TLS within this paradigm. Ideally, a simplified TLS API designed around the POSIX socket API would merely add TLS as a new parameter value for the protocol (`IPPROTO_TLS`). Subsequent calls to POSIX socket functions such as `connect`, `send`, and `recv` would then perform the TLS handshake, encrypt and transmit data, and receive and decrypt data respectively, based on the TLS protocol. Our design goals are as follows:

1. Enable developers to use TLS through the existing set of functions provided by the POSIX socket API, without adding any new functions or changing of function signatures. Modifications to the API are acceptable only in the form of new *values* for existing parameters. This enables us to provide an API that is already well-known to network programmers and implemented by many existing programming languages, which simplifies both automatic and manual porting to the SSA.
2. Support direct administrator control over the parameters and settings for TLS connections made by the SSA. Applications should be able to increase, but not decrease, the security preferred by the administrator.
3. Export a minimal set of TLS options to applications that allow general TLS use and drastically reduce the amount of TLS functions in contemporary TLS APIs.
4. Facilitate the adoption of the SSA by other programming languages, easing the security burden on language implementations and providing broader security control to administrators.

### 3.5 OpenSSL Analysis

In the pursuit of our goals, we first gather design recommendations and assess the feasibility of our approach by analyzing the OpenSSL API and how it is used by popular software packages. We explore what functionality should be present in the SSA and how to distill the 504 TLS-related OpenSSL symbols (e.g., functions, macros) to the handful provided by the POSIX socket interface. We limit our analysis to the features exported by `libssl`, the component of OpenSSL responsible for TLS functionality. With few exceptions, `libcrypto`, which supports generic cryptographic activities, is out of the scope of our study. GnuTLS and other libraries could also have been explored, but we choose OpenSSL due to its popularity and expansive feature set, leaving the assessment of other libraries to future work. For the results outlined, we analyzed OpenSSL 1.0.2 and software packages from Ubuntu 16.04. A full listing of our methods and results for our analysis of `libssl` is located at `owntrust.org`.

We collected the source code for all standard Ubuntu repository software packages that directly depend on `libssl`. We then filtered the resulting 882 packages for those using C/C++, leaving 410 packages for our analysis of direct use of `libssl`. Of these, 276 have TLS server functionality and 340 have TLS client functionality (248 have both). Note that packages using other languages may depend on OpenSSL by utilizing one of the packages in our analysis. We analyzed the source code of each package in our derived set in the context of its use of the symbols exported by `libssl`.

To obtain a comprehensive list of functionality offered by `libssl`, we extracted the symbols (e.g., functions, constants) it exports to applications. We also augmented this list of 323 symbols by recursively adding preprocessor macros that use already-identified symbols. This resulted in a cumulative list of 504 unique API symbols that developers can use when interfacing with OpenSSL's `libssl`. We then cataloged the behavior and uses of each of these symbols using descriptions in the official API documentation, in cases where such entries existed. Manual inspection of source code and unofficial third-party documentations were used to catalog symbols not present in the official documentation. We categorized each of

| Category | Symbols | Uses |
|---|---|---|
| TLS Functionality | | |
| Version selection | 29 | 1306 |
| Cipher suite selection | 39 | 1467 |
| Extension management | 68 | 597 |
| Certificate/Key management | 73 | 2083 |
| Certificate/Key validation | 51 | 3164 |
| Session management | 61 | 1155 |
| Configuration | 19 | 1337 |
| Other | | |
| Allocation | 33 | 6087 |
| Connection management | 41 | 5228 |
| Miscellaneous | 64 | 1468 |
| Instrumentation | 26 | 232 |

Table 3.1: Breakdown of OpenSSL's `libssl` symbols.

the symbols into the groups shown in Table 3.1. Our selection of packages made a total of 24,124 calls to the `libssl` API.

The resulting categories are of two types: those that are used for specifying behavior of the TLS protocol itself (e.g., symbols that indicate which TLS version to use, or how to validate a certificate), and those that relate specifically to OpenSSL's implementation (e.g., symbols used to allocate and free OpenSSL structures, options to turn on bug workarounds). For each category, we employed both automated static code analysis techniques, using Joern [93], and manual inspection to understand the use cases for each of its symbols.

Immediately we found that 170 of the 504 API symbols are not used by any application in our analysis. Despite this, we manually inspected every symbol in the API to determine whether they offered an important use case for the SSA. The highlights of our findings for select categories are as follows.

### 3.5.1 Version Selection

OpenSSL allows developers to specify the versions of TLS which their connections should use, and retrieve this information. Of calls that set a version, 459 (54%) are functions prefixed

with `SSLv23`, which default to the latest TLS version supported by OpenSSL, but also allow fallback to supported previous versions. The OpenSSL documentation indicates that these functions are preferred [34]. Of the 388 (68%) calls that indicate a singular TLS version to use, only 60 (15%) use the latest version of TLS (1.2), and 83 (21%) specify the use of the vulnerable SSL 3.0. Another 190 (49%) directly specify the use of TLS 1.0, through the use of `TLSv1_method` settings. Our inspection of source code comments surrounding these uses suggest that many developers erroneously believe that it selects the latest TLS version. We also found that many uses of version selection functions are determined by compile-time settings supplied by package maintainers and system administrators.

In aggregate, these version selection behaviors suggest that overwhelmingly developers want the system to select the version for them, directly or indirectly, or are adopting lower versions erroneously. We therefore recommend that the SSA use the latest uncompromised TLS versions by default, and that deviation from this be controlled by the system administrator.

### 3.5.2 Cipher Suite Selection

In our dataset, 221 (54%) packages contain code that sets the ciphers used by OpenSSL directly, using the `*_set_cipher_list` functions. Due to limitations in how Joern performs static analysis, we are not able to determine all of the parameter values provided to these functions. However, a sample of applications with hard-coded ciphers suggests some bad practice. Of note are the uses of `eNULL` (5), `NULL` (10), `COMPLEMENTOFALL` (3), `RC4` (2), and `MD5` (1), all of which enable vulnerable ciphers or enable the null cipher, which offers no encryption at all. We manually analyzed an additional sample of packages and found that many adopt default settings or retrieve their cipher suite lists dynamically from environment variables and configuration files.

Our analysis indicates that, like with version selection, developers want to let the system select cipher suites for them, and that those who choose to hardcode behaviors often make mistakes. We thus recommend that allowed cipher suites be set by the system

administrator. The SSA could allow applications to further limit cipher suites, but should not let them request suites that are not allowed by the administrator.

### 3.5.3 Extension Management

OpenSSL exports explicit control of ten TLS extensions through functions in the extension management category. Only two extensions are used somewhat regularly – Server Name Indication (SNI), in 77 (19%) applications, and Next Protocol Negotiation (NPN) and its successor Application-Layer Protocol Negotiation (ALPN), in 60 (15%) applications. Five other extensions–including Online Certificate Status Protocol (OCSP)– are used much less often, and Heartbeats, PRF, Serverinfo, and Supported Curves are not used at all.

Our observation is that many extensions should be configured by the system administrator. For example, SNI and OCSP could be enabled system-wide so that all applications use them. In addition, there are relatively few cases where developers need to supply configuration for an extension, such as a hostname with SNI or a list of protocols with ALPN. We therefore recommend that the SSA implement extensions on behalf of the application and expose an interface to developers for supplying configuration information.

### 3.5.4 Certificate/Key Management

Of the 73 API functions used for managing keys and certificates, 39 (54%) are unused. Another 17 (23%) are used by less than five software packages. The remaining functions are used heavily, with a combined call count of 2083 from hundreds of distinct packages. Most of these are used to either specify a certificate or private key for the TLS connection. However, one is used to verify that a given private key corresponds to a particular certificate, and two are used to provide decryption passphrases to unlock private keys.

Given that most functions in this category are unused, and that all but three of those that are used are for specifying the locations of certificates and private keys, we recommend the SSA have simplified options for supplying private key and certificate data. These options

should take both chains and leaf certificates as input, in keeping with recommendations in the OpenSSL documentation. Additionally, the SSA can check whether a supplied key is valid for supplied certificates on behalf of the developer, removing the need for developers to check this themselves, reporting relevant errors through return values of key assignment functionality.

### 3.5.5 Certificate Validation

Under TLS, failure to properly validate a certificate presented by the other endpoint undermines authentication guarantees. Previous research has shown that developers often make mistakes with validation [14, 31, 35]. Our analysis indicates that the certificate validation functions in OpenSSL are heavily used, but confirms that developers continue to make mistakes. We found that 6 packages disable validation entirely and specify no callback for custom validation, indicating the presence of a man-in-the-middle vulnerability. We have notified the relevant developers of these problems. A total of 7 packages use `SSL_get_verify_result`, but neglect to ensure `SSL_get_peer_certificate` returns a valid certificate. Neglecting this call is documented as a bug in the OpenSSL documentation, because receiving no certificate results in a success return value.

Recent work has described the benefits of handling verification in an application-independent manner and under the control of administrator preferences [10, 66]. Given this work and the poor track record of applications, we recommend that validation be performed by the SSA, which should implement administrator preferences and provide secure defaults. This includes the employ of strengthening technologies such as OSCP [81], CRLs [16], etc. We make this recommendation with one caveat: if an application would like to validate a certificate based on a hard-coded set or its own root store, then it can supply a set of trusted certificates to the SSA.

### 3.5.6 Session Management

Performing the TLS handshake requires multiple round trips, which can be relatively expensive for latency-sensitive applications. Session caching alleviates this by storing TLS session data for resumption during an abbreviated handshake. Most of the analyzed packages, 299 (73%), do not make any changes to the default session caching mechanisms of OpenSSL. Within the other 27%, the most common modification is to simply turn caching off entirely. The remaining uses disable individual caching features or are calls to explicitly retain default settings. There are 31 packages that implement custom session cache handling. Manual inspection of these packages found this was used for logging and to pass session data to other processes, presumably to support load balancing for servers.

We recommend that session caching be implemented by the SSA, relieving developers of this burden, with options for developers to disable caching and customize session TTLs. Because it operates as an OS service, the SSA is uniquely positioned to allow sharing of session state between processes of the same application. This could be further adapted to support session sharing between instances of an application on different machines.

### 3.5.7 Configuration

OpenSSL provides configuration of various options that control the behavior of TLS connections, along with modes that allow fine-tuning the TLS implementation, such as indicating when internal buffers should be released or whether to automatically perform renegotiation. Most calls in this category, 830 (62%), are used to adjust options. The four most-used options disable vulnerable TLS features and older versions (e.g., compression, SSLv2, SSLv3), and enable all bug workarounds (for interoperability with other TLS implementations). An additional 337 (25%) calls in this category set various modes. Of these, 138 (41%) set a flag that makes I/O operations on a socket block if the handshake has not yet completed, 189 (56%) set flags that modify the `SSL_write` function to behave more like `write`, and 47 (14%) use a flag that reduces the memory footprint of idle TLS connections. Also present

are 32 calls (2%) to functions that change how many bytes OpenSSL reads during receive operations. Through manual inspection we find that many of these configurations are set by compilation parameters, suggesting that many developers are leaving these decisions to administrators already.

Given that the uses of this category are primarily bug workarounds and restricting the use of outdated protocols, and that many of these are already set through compilation flags, we recommend leaving such configurations to the administrator. Software updates can apply bug workarounds and disable vulnerable protocols in one location, deploying them to all applications automatically. Modes and other configuration settings in this category tend to control subtleties of read and write operations. Under the SSA, I/O semantics are largely determined by the existing POSIX socket standard, so we ignore them.

### 3.5.8 Non-TLS Protocol Specific Functions

The remaining categories consist of functions not applicable to the SSA or those trivially mapped to it. The allocation category contains functions such as `SSL_library_init` and `SSL_free`, whose existence is obviated by the existence of the SSA because all relevant memory allocation and freeing is performed as part of calls such as `socket` and `close`. The connection management category contains functions that perform connection and I/O operations on sockets. All of these have direct counterparts within the POSIX socket API, or have combinations of symbols that emulate the behavior, such as `SSL_connect` (`connect`), and `SSL_Peek` (`recv` with `MSG_PEEK` flag). Another example is that of `SSL_get_error`, which when called returns a value similar to `errno`. These functions should therefore be mapped to their POSIX counterparts for the SSA. The instrumentation and miscellaneous categories contain functionality that monitors raw TLS messages, extracts information from internal data structures, is scheduled for deprecation, etc.

## 3.6 The Secure Socket API

We designed the SSA using lessons learned from our study of `libssl` and its usage. The SSA is responsible for automatic management of every TLS category discussed in the previous section, including automatic selection of TLS versions, cipher suites, and extensions. It also performs automatic session management and automatic validation of certificates. By using standard network send and receive functions, the SSA automatically and transparently performs encryption and decryption of data for applications, passing relevant errors through `errno`. All of these are subject to a system configuration policy with secure defaults, with customization abilities exported to system administrators and developers. Administrators set global policy (and can set policy for individual applications), while developers can choose to further restrict security. Developers can increase security, but cannot decrease it.

### 3.6.1 Usage

Under the Secure Socket API, all TLS functionality is built directly into the POSIX socket API. The POSIX socket API was derived from Berkeley sockets and is meant to be portable and extensible, supporting a variety of network communication protocols. As a result, TLS fits nicely within this framework, with support for all salient operations integrated into existing functions without the need for additional parameters, pursuant to our first design goal. When creating a socket, developers select TLS by specifying the protocol as `IPPROTO_TLS`. Data is sent and received through the socket using standard functions such as `send` and `recv`, which will be encrypted and decrypted using TLS, just as network programmers expect their data to be placed inside and removed from TCP segments under `IPPROTO_TCP`. To transparently employ TLS in this fashion, other functions of the POSIX socket API have specialized TLS behaviors under `IPPROTO_TLS` as well. Table 3.2 contains a brief description of the POSIX socket API functions with the specific behaviors they adopt under TLS.

To offer concrete examples of SSA utilization, we also present code for a simple client and server in Figure 3.1. Both the client and the server create a socket with the `IPPROTO_TLS`

| POSIX Function | General Behavior | Behavior under IPPROTO_TLS |
|---|---|---|
| `socket` | Create an endpoint for communication utilizing the given protocol family, type, and optionally a specific protocol. | Create an endpoint for TLS communication, which utilizes TCP for its transport protocol if the `type` parameter is `SOCK_STREAM` and uses DTLS over UDP if `type` is `SOCK_DGRAM`. |
| `connect` | Connect the socket to the address specified by the `addr` parameter for stream protocols, or indicate a destination address for subsequent transmissions for datagram protocols. | Perform a connection for the underlying transport protocol if applicable (e.g., TCP handshake), and perform the TLS handshake (client-side) with the specified remote address. Certificate and hostname validation is performed according to administrator and as optionally specified by the application via `setsockopt`. |
| `bind` | Bind the socket to a given local address. | No TLS-specific behavior. |
| `listen` | Mark a connection-based socket (e.g., `SOCK_STREAM`) as a passive socket to be used for accepting incoming connections. | No TLS-specific behavior. |
| `accept` | Retrieve connection request from the pending connections of a listening socket and create a new socket descriptor for interactions with the remote endpoint. | Retrieve a connection request from the pending connections, perform the TLS handshake (server-side) with the remote endpoint, and create a new descriptor for interactions with the remote endpoint. |
| `send`, `sendto`, etc. | Transmit data to a remote endpoint. | Encrypt and transmit data to a remote endpoint. |
| `recv`, `recvfrom`, etc. | Receive data from a remote endpoint. | Receive and decrypt data from a remote endpoint. |
| `shutdown` | Perform full or partial tear-down of connection, based on the `how` parameter. | Send a TLS close notify. |
| `close` | Close a socket, perform connection teardown if there are no remaining references to socket. | Close a socket, send a TLS close notify, and tear-down connection, if applicable. |
| `select`, `poll`, etc. | Wait for one or more descriptors to become ready for I/O operations. | No TLS-specific behavior. |
| `setsockopt` | Manipulate options associated with a socket, assigning values to specific options for multiple protocol levels of the OSI stack. | Manipulate TLS specific options when the `level` parameter is `IPPROTO_TLS`, such as specifying a certificate or private key to associate with the socket. Other `level` values interact with the socket according to their existing semantics. |
| `getsockopt` | Retrieve a value associated with an option from a socket, specified by the `level` and `option_name` parameters. | For a `level` value of `IPPROTO_TLS`, retrieve TLS-specific option values. Other `level` values interact with the socket according to their existing semantics. |

Table 3.2: Brief descriptions of the behavior of POSIX socket functions generally and under `IPPROTO_TLS` specifically. General behavior is paraphrased from relevant manpages.

```
/* Use hostname address family */
struct sockaddr_host addr;
addr.sin_family = AF_HOSTNAME;
strcpy(addr.sin_addr.name, "www.example.com");
addr.sin_port = htons(443);

/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
/* TLS Handshake (verification done for us) */
connect(fd, &addr, sizeof(addr));

/* Hardcoded HTTP request */
char http_request[] = "GET / HTTP/1.1\r\nhost: www.example.com\r\n\r\n";
char http_response[2048];
memset(http_response, 0, 2048);
/* Send HTTP request encrypted with TLS */
send(fd,http_request,sizeof(http_request)-1,0);
/* Receive decrypted response */
recv(fd, http_response, 2047, 0);
/* Shutdown TLS connection and socket */
close(fd);
/* Print (possibly partial) response */
printf("Received:\n%s", http_response);
```

(a) A simple HTTPS client example under the SSA. Error checks and some trivial code are removed for brevity. Alternatively, the client could have used the `TLS_REMOTE_HOSTNAME` option with `setsockopt` to indicate the hostname, and called `connect` using traditional `AF_INET` or `AF_INET6` address families.

```
/* Use standard IPv4 address */
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
/* We want to listen on port 443 */
addr.sin_port = htons(443);

/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);
/* Bind to local address and port */
bind(fd, &addr, sizeof(addr));
/* Assign certificate chain */
setsockopt(fd, IPPROTO_TLS, TLS_CERTIFICATE_CHAIN, CERT_FILE, sizeof(CERT_FILE));
/* Assign private key */
setsockopt(fd, IPPROTO_TLS, TLS_PRIVATE_KEY, KEY_FILE, sizeof(KEY_FILE));
listen(fd, SOMAXCONN);

while (1) {
  struct sockaddr_storage addr;
  socklen_t addr_len = sizeof(addr);
  /* Accept new client and do TLS handshake
  using cert and keys provided */
  int c_fd = accept(fd, &addr, &addr_len);
  /* Receive decrypted request */
  recv(c_fd, request, BUFFER_SIZE, 0);
  handle_req(request, response);
  /* Send encrypted response */
  send(c_fd, response, BUFFER_SIZE, 0);
  close(c_fd);
}
```

(b) A simple server example under the SSA. Error checks and some trivial code are removed for brevity.

Figure 3.1: Code examples for applications using the SSA.

protocol. The client uses the standard `connect` function to connect to the remote host, also employing the `AF_HOSTNAME` address family to indicate to which hostname it wishes to connect. The client `send`s a plaintext HTTP request to the selected server, which is then encrypted by the SSA before transmission. The response received is also decrypted by the SSA before placing it into the buffer provided to `recv`.

In the server case, the application calls `bind` to give itself a source address of 0.0.0.0 (`INADDR_ANY`) on port 443. Before it calls `listen`, it uses two calls to `setsockopt` to provide the location of its private key and certificate chain file to be used for authenticating itself to clients during the TLS handshake. After the listening descriptor is established, the server then iteratively handles requests from incoming client connections, and the SSA performs a handshake with clients transparently using the provided options. As with the client case, calls to `send` and `recv` have their data encrypted and decrypted in accordance with the TLS session, before they are delivered to relevant destinations.

### 3.6.2 Administrator Options

Our second design goal is to enable administrator control over TLS parameters set by the SSA. Administrators gain this control through a protected configuration file, which exports the following options:

- **TLS Version:** Select which TLS versions to enable, in order of preference (default: TLS 1.2, TLS 1.1, TLS 1.0).
- **Cipher Suites:** Select which cipher suites to enable, in order of preference (vulnerable ciphers are disabled by default).
- **Certificate Validation:** Select active certificate validation mechanisms and strengthening technologies. We cover this in more detail at the end of this section.
- **Honor Application Validation:** Specify whether to honor validation against root stores supplied by applications (default: true).

- **Enabled Extensions:** Specify names of extensions to employ (e.g., "ALPN").
- **Session Caching:** Configure session cache information (TTL, size, location).
- **Default Paths:** Specify default paths for the private keys and certificates to employ when developers do not supply them.

#### 3.6.2.1 Application Profiles

The settings mentioned are applied to all TLS connections made with the SSA on the machine. However, additional configuration profiles can be created or installed by the administrator for specific applications that override the global settings. The SSA enforces global TLS policy for any application, unless a configuration profile for that specific application is present, in which case it enforces the settings from the application-specific profile. We do this in a fashion similar to the application-specific profiles of AppArmor [91], the mandatory access control module used by Ubuntu and other Linux distributions. Under AppArmor, application-specific access control policy is defined in a textual configuration file, which specifies the target application using the file system path to the executable of the application. When the application is run, AppArmor uses the rules in the custom profile when enforcing access control policy. Ubuntu ships with AppArmor profiles for a variety of common applications. Administrators can create their own profiles or customize those supplied by their OS vendor. We adopt a similar scheme, in which TLS configuration can be tailored to specific applications using custom SSA configuration profiles. These application profiles can be distributed by OS vendors, application developers, and third parties, or created by administrators. In any case, administrators are free to modify any configuration to match their policies.

#### 3.6.2.2 Certificate Validation

Special care is given to certificate validation as it is complex and commonly misused. In an effort to maximize security and the flexibility available to administrators, the SSA allows administrators to select between standard validation and TrustBase [66]. Under

| IPPROTO_TLS socket option | Purpose |
|---|---|
| TLS_REMOTE_HOSTNAME | Used to indicate the hostname of the remote host. This option will cause the SSA to use the Server Name Indication in the TLS Client Hello message, and also use the specified hostname to verify the certificate in the TLS handshake. Use of the AF_HOSTNAME address type in connect will set this option automatically. |
| TLS_HOSTNAME | Used to specify and retrieve the hostname of the local socket. Servers can use this option to multiplex incoming connections from clients requesting different hostnames (e.g., hosting multiple HTTPS sites on one port). |
| TLS_CERTIFICATE_CHAIN | Used to indicate the certificate (or chain of certificates) to be used for the TLS handshake. This option can be used by both servers and clients. A single certificate may be used if there are no intermediate certificates to be used for the connection. The value itself can be sent either as a path to a certificate file or an array of bytes, in PEM format. This option can be set multiple times to allow a server to use multiple certificates depending on the requests of the client. |
| TLS_PRIVATE_KEY | Used to indicate the private key associated with a previously indicated certificate. The value of this option can either be a path to a key file or an array of bytes, in PEM format. The SSA will report an error if the provided key does not match a provided certificate. |
| TLS_TRUSTED_PEER_CERTIFICATES | Used to indicate one or more certificates to be a trust store for validating certificates sent by the remote peer. These can be leaf certificates that directly match the peer certificate and/or those that directly or indirectly sign the peer certificate. Note that in the presence or absence of this option, peer certificates are still validated according to system policy. |
| TLS_ALPN | Used to indicate a list of IANA-registered protocols for Application-Layer Protocol Negotiation (e.g., HTTP/2), in descending order of preference. This option can be fetched after connect/accept to determine the selected protocol. |
| TLS_SESSION_TTL | Request that the SSA expire sessions after the given number of seconds. A value of zero disables session caching entirely. |
| TLS_DISABLE_CIPHER | Request that the underlying TLS connection not use the specified cipher. |
| TLS_PEER_IDENTITY | Request the identity of remote peer as indicated by the peer's certificate. |
| TLS_PEER_CERTIFICATE_CHAIN | Request the remote peer's certificate chain in PEM format for custom inspection. |

Table 3.3: Sample of socket options at the IPPROTO_TLS level

standard validation, traditional certificate validation will be performed. This includes some additional checks made by strengthening technologies, such as revocation checks, where available. TrustBase is available for administrators who wish to have finer-grained control over validation, or who wish to employ more exotic validation mechanisms. Under TrustBase, administrators can employ multiple validation strategies, and use them simultaneously with various aggregation policies. For example, using TrustBase, we have deployed validation strategies consisting of combinations of standard validation, OCSP checking [81], Google CRLset checking [73], certificate pinning, and DANE [39]. Additional validation mechanisms not listed can also be used, such as notary-based validation, through the TrustBase plugin API.

### 3.6.3 Developer Options and Use Cases

The `setsockopt` and `getsockopt` POSIX functions provide a means to support additional settings in cases where a protocol offers more functionality than can be expressed by the limited set of principal functions. Under Linux, 34 TCP-specific socket options exist to customize protocol behavior. For example, the `TCP_MAXSEG` option allows applications to specify the maximum segment size for outgoing TCP packets. Arbitrary data can be transferred to and from the API implementation using `setsockopt` and `getsockopt`, because they take a generic pointer and a data length (in bytes) as parameters, along with an `optname` constant identifier. Adding a new option can be done by merely defining a new `optname` constant to represent it, and adding appropriate handling code to the implementation of `setsockopt` and `getsockopt`.

In accordance with this standard, the SSA adds a few options for `IPPROTO_TLS`. These options and their uses are described in Table 3.3. These reflect a minimal set of recommendations gathered from our analysis of existing TLS use by applications, reflecting our third design goal. This set can easily be expanded to include other options as their use

cases are explored and justified. We caution against adding to this list ad nauseam, as it may undermine the simplicity with which developers interact with the SSA.

In many cases, a developer writing TLS client code only needs to write or change a few lines of code to create a secure connection. The developer simply uses `IPPROTO_TLS` as the third parameter of their call to `socket` and then calls `setsockopt` with the `TLS_REMOTE_HOSTNAME` option to provide a destination hostname. Use of this option allows SSA to automatically include the SNI extension and properly validate the hostname for a certificate offered by a server. To streamline this process, we add a new `sockaddr` type, `AF_HOSTNAME`, which can be supplied to `connect`. Some languages, such as Python, have already made this change to their analog of `connect`, allowing hostnames to be provided in place of IP addresses. When supplied with a hostname address type, the `connect` function will perform the necessary host lookup and perform a TLS handshake with the resulting address, also using the provided hostname for certificate validation and the SNI extension. This also obviates the need for developers to explicitly call `gethostbyname` or `getaddrinfo` for hostname lookups, which further simplifies their code.

The SSA enables a useful split between administrator and developer responsibilities for secure servers. An administrator can use software from *Let's Encrypt* to automatically obtain certificates for the hostnames associated with a given machine, and associate those certificates (and keys) with an SSA profile for the application. All the developer needs to do to create a secure server is to specify `IPPROTO_TLS` in their call to `socket`, and then bind to all interfaces on a given machine. When incoming clients specify a hostname with SNI, the SSA automatically supplies the appropriate certificate for the hostname. If an incoming socket does not use SNI, then the SSA defaults to the first certificate listed in its configuration. If the developer wishes to bind to a particular hostname, then they may use `setsockopt` with the `TLS_HOSTNAME` option on their listening socket.

| Program | LOC Modified | LOC removed | Familiar with code | Time Taken |
|---|---|---|---|---|
| wget | 15 | 1,020 | No | 5 Hrs. |
| lighttpd | 8 | 2,063 | No | 5 Hrs. |
| ws-event | 5 | 0 | Yes | 5 Min. |
| netcat | 5 | 0 | No | 10 Min. |

Table 3.4: Summary of code changes required to port a sample of applications to use the SSA. wget and lighttpd used existing TLS libraries, ws-event and netcat were not originally TLS-enabled. LOC = Lines of Code

The options listed in Table 3.3 are useful primarily in special cases, such as for client certificate pinning, or specifying a particular certificate and private key to use in the TLS handshake.

### 3.6.4 Porting Applications to the SSA

To obtain metrics on porting applications to use the SSA, we modified the source code of four network programs. Two of these already used OpenSSL for their TLS functionality, and two were not built to use TLS at all. Table 3.4 summarizes the results of these efforts.

We modified the command-line `wget` web client to use the SSA for its secure connections. Normally, `wget` links with either GnuTLS or OpenSSL for TLS support, based on compilation configuration. Our modifications required only 15 lines of source code. These changes involved using `IPPROTO_TLS` in the `socket` call when the URL scheme was secure (e.g., HTTPS, FTPS) and then assigning the appropriate hostname to the socket, using `setsockopt` with the `TLS_REMOTE_HOSTNAME` option. The resulting binary could then be compiled without linking with either GnuTLS or OpenSSL, removing 1,020 lines of OpenSSL-using code and allowing the administrator to dictate the parameters of TLS connections made. This modification was made in five hours by a programmer with no prior experience with `wget`'s source code or OpenSSL, but who had a working knowledge of C and POSIX sockets.

We also modified `lighttpd`, a light-weight event-driven TLS webserver, to use the SSA instead of OpenSSL. This required only the modification of four lines of code, which

merely specified `IPPROTO_TLS` in places where sockets were created. We also made optional calls to `setsockopt` to specify the private key and certificate chain (and check errors), with an additional four lines of code. We removed 2,063 lines of code used for interfacing with OpenSSL. These software packages were then tested to ensure that they functioned properly and used the TLS settings enforced by the SSA. This modification was made in five hours by another individual with no prior experience with `lighttpd`'s source code or OpenSSL, but who had a working knowledge of C and POSIX sockets. In porting this and `wget`, most of the time spent was used to become familiar with the source code and remove OpenSSL calls.

We also modified two applications that did not previously use TLS, an in-house webserver and the `netcat` utility. The webserver required modifying only one line of code—the call to `socket` to use `IPPROTO_TLS` on its listening socket. Under these circumstances, the certificate and private key used are from the SSA configuration. However, these can be specified by the application with another four lines of code to set the private key and certificate chain and check for corresponding errors. In total, this TLS upgrade required less than five minutes. The TLS upgrade for `netcat` for both server and client connections required modifying five lines of code and was accomplished in under ten minutes, with the developer not being familiar with the code beforehand.

These efforts suggest that porting insecure programs to use the SSA can be accomplished quickly and that porting OpenSSL-using code to use the SSA can be relatively easy, even without prior knowledge of the codebase.

### 3.6.5 Language Support

One of the benefits of using the POSIX socket API as the basis for the SSA is that it is easy to provide SSA support to a variety of languages, which is in line with our fourth design goal. This benefit accrues if an implementation of the SSA instruments the POSIX socket functionality in the kernel through the system call interface, which all network-using languages already rely upon. Any language that uses the network must interface with network

system calls, either directly through machine instructions or indirectly by wrapping another language's implementation. Therefore, given an implementation in the kernel, it is trivial to add SSA support to other languages that have networking support. We describe how our implementation accomplishes this in Section 3.7.

To illustrate this benefit, we have added SSA support to three additional languages beyond C/C++: Python, PHP, and Go. We chose these languages due to the fact that each uses a different approach for requesting network communication from the kernel. The modifications required to provide SSA support for these languages are as follows.

- **Python:** The reference implementation of the Python interpreter is written in C and uses the POSIX socket API for networking support. Adding SSA support to Python required modification of `socketmodule.c`, which was done by merely adding SSA constants (i.e., `IPPROTO_TLS` and option values for `setsockopt`/`getsockopt`.)
- **PHP:** The common PHP interpreter passes parameters from its socket library directly to its system call implementation. This means that modification of the interpreter isn't strictly necessary to support the SSA; applications can supply constants themselves to use for `IPPROTO_TLS` and the values for options. Adding these values to the interpreter required the definition of SSA constants.
- **Go:** Go is a compiled language and thus uses system calls directly. Adding SSA support to Go merely required adding a new constant, "tls", and an associated numerical value, to the `net` package of the language. Go also provides functions to interface with the `setsockopt` and `getsockopt` system calls (e.g., `SetsockoptInt`), which allow light-weight wrappers of options (e.g., `setNoDelay`) to be made. Adding an SSA option function in a similar fashion requires only 2-3 lines of Go code. With these changes to the Go standard library, application developers can create a TLS socket by specifying "tls" when they `Dial` a connection. To test and demonstrate these changes, we ported Caddy [40], a popular Go-based HTTP/2 webserver, to the SSA for its Internet connections.

Together these efforts illustrate the ease of adding SSA support to various languages. The majority of the work required is to define a few constants for existing system calls or their wrappers.

### 3.6.6 TLS 1.3 0-RTT

TLS 1.3 provides a "0-RTT" mode, which allows clients to resume an existing TLS session and provide application data with a single TLS message. Used incorrectly this feature may be vulnerable to replay attacks, but nonetheless offers a significant latency benefit when employed correctly. The 0-RTT mode is unique in that it combines connect and send operations. Fortunately, the socket API has already been adapted to deal with previous protocol changes that combined these operations, such as TCP Fast Open (TFO). TFO is supported by clients via the `sendto` (or `sendmsg`) function with the `MSG_FASTOPEN` flag. This allows the developer to specify a destination for the connection and data to send using a single function. TFO is supported by servers by setting the `TCP_FASTOPEN` option on their listening socket. Alternatively, the `TCP_FASTOPEN_CONNECT` option allows TFO client functionality using a lazy `connect` and subsequent `send`. The SSA can support TLS 1.3 0-RTT using similar mechanisms, leveraging `sendto` with a flag or the `TLS_ORTT` socket option.

## 3.7 Implementation Details

We have developed a loadable Linux kernel module that implements the Secure Socket API. Source code is available at `owntrust.org`.

A high-level view of a typical network application using a security library for TLS is shown in Figure 3.2. The application links to the security library, such as OpenSSL or GnuTLS, and then uses the POSIX Socket API to communicate with the network subsystem in the kernel, typically using a TCP socket.

A corresponding diagram, shown in Figure 3.3, illustrates how our implementation of the SSA compares to this normal usage. We split our SSA implementation into two

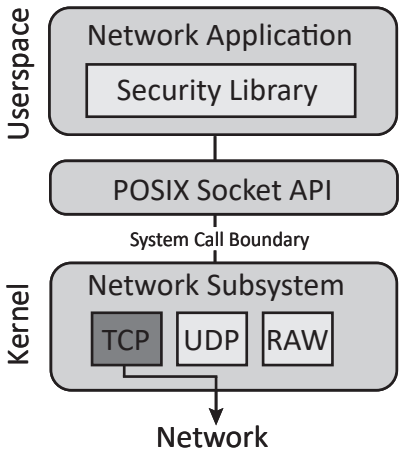



Figure 3.2: Data flow for traditional TLS library by network applications. The application shown is using TCP.

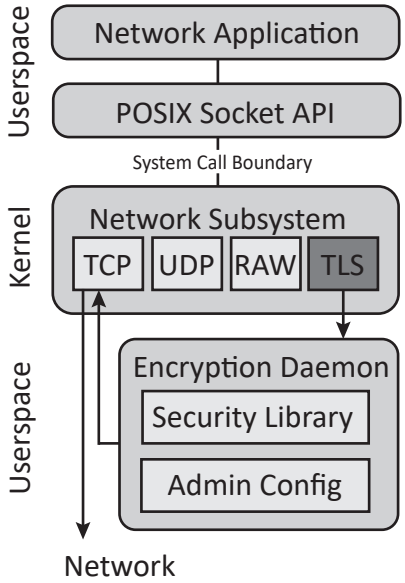



Figure 3.3: Data flow for SSA usage by network applications. The application shown is using the TLS (which uses TCP internally for connection-based `SOCK_STREAM` sockets).

parts: a kernel component and a user space encryption daemon accessible only to the kernel component. At a high-level, the kernel component is responsible for registering all `IPPROTO_TLS` functionality with the kernel and maintaining state for each TLS socket. The kernel component offloads the tasks of encryption and decryption to an encryption daemon, which uses OpenSSL and obeys administrator preferences.

Note that our prototype implementation moves the use of a security library to the encryption daemon. The application interacts only with the POSIX Socket API, as described in Section 3.6, and the encryption daemon establishes TLS connections, encrypts and decrypts data, implements TLS extensions, and so forth. The daemon uses administrator configuration to choose which TLS versions, cipher suites, and extensions to support. It should be noted that while modern TLS libraries are complicated and difficult to use, libraries like OpenSSL have a strong deployment base and a large history of testing and bug fixing that are difficult to rival. Our prototype implementation leverages this by calling the OpenSSL library on behalf of applications. Writing TLS functionality in kernel code (i.e. not user space) is an undertaking outside the scope of this work, and one which should involve extensive participation from the security community.

### 3.7.1 Basic Operation

The Linux kernel allows the same network system calls to handle different protocols by storing pointers to the kernel functions associated with a given protocol inside generalized socket objects. The kernel component of our SSA implementation supplies its own functions for TLS behavior, using the kernel to associate these functions with all sockets created using `IPPROTO_TLS`. The supplied functions are then invoked when a user application invokes a corresponding POSIX socket call on a TLS socket, through the system call interface.

When an SSA-using application invokes an I/O operation on a TLS socket, the kernel component transfers the plaintext application data to the user space daemon for encryption, and the encrypted data are then transmitted to the intended remote endpoint. In the reverse

direction, encrypted data from the remote endpoint are decrypted by the daemon and then sent to the kernel to be delivered to the client application. The user space encryption daemon is a multi-process, event-driven service that interacts with the OpenSSL library to perform TLS operations. The kernel load balances TLS connections across active daemon processes to take advantage of the parallelism provided by multicore CPUs.

To accomplish its tasks, the kernel component must inform the daemon of important events triggered by application system calls. A selection of these events and their descriptions are as follows:

- **Socket creation** When a TLS socket is created by an application, the kernel informs the daemon that it must create a corresponding socket of the appropriate transport protocol, known as the *external* socket. Unknown to the application, this external socket is used for direct communication with the intended remote host. The TLS socket created by the application, known as the *internal* socket, is used to transfer plaintext data to and from the daemon.
- **Binding** After TLS socket creation, an application may choose to call `bind` on that socket, requesting that the socket use the specified source address and port. Since the daemon interfaces directly with remote hosts, the kernel directs the daemon to `bind` on the external socket.
- **Connecting** When an application calls `connect`, the kernel informs the daemon to connect its external socket to the address specified by the application, and then connects the internal socket to the daemon.
- **Listening** Server applications may call `listen` on their socket. In this case, the kernel informs the daemon of this action, and both the external and internal socket are placed into listening mode.
- **Socket options** Throughout a TLS socket's lifetime, an application may wish to use `setsockopt` or `getsockopt` to assign and retrieve information about various socket

behaviors. Notification of these options and their values is provided by the kernel to the daemon. Setting socket options with level `IPPROTO_TLS` are directly handled by the daemon, which appropriately sets and retrieves TLS state depending on the requested option. Setting options at other levels, such as `IPPROTO_TCP` or `SOL_SOCKET`, are performed on both internal and external sockets, where appropriate.

Handling of these application requests using the encryption daemon is done in a manner invisible to the application. Special care is given to error returns and state to guarantee consistency between external and internal sockets. For example, if the daemon fails to connect to a specified remote host, the corresponding error code is sent back to the application, and the kernel does not connect the internal socket to the the daemon, maintaining both sockets in an unconnected state and informing the application of real errors.

When the daemon receives a certificate from a remote peer, it validates that certificate based on administrator preferences. The administrator can employ traditional certificate validation checks using a certificate trust store and the hostname provided by the application through `TLS_REMOTE_HOSTNAME`. Remote TLS client connections are authenticated using the trusted peer certificates, optionally supplied by a server application, as a trust store. In addition to, or replacement of these methods, administrators can defer validation to TrustBase [66], which offers multiple coexisting certificate validation strategies.

Creating an internal socket between applications and the daemon provides natural support for existing socket I/O and polling operations. Read and write operations can use their existing kernel implementations with no modification, and event notifications from the kernel through the use of `select`, `poll`, and `epoll` are handled automatically.

### 3.7.2 Performance

We performed stress tests to ensure that the encryption daemon could feasibly act as an encryption proxy for numerous applications simultaneously. We wrote two client applications, one using the SSA and the other using OpenSSL, that download a 1MB file over HTTPS using

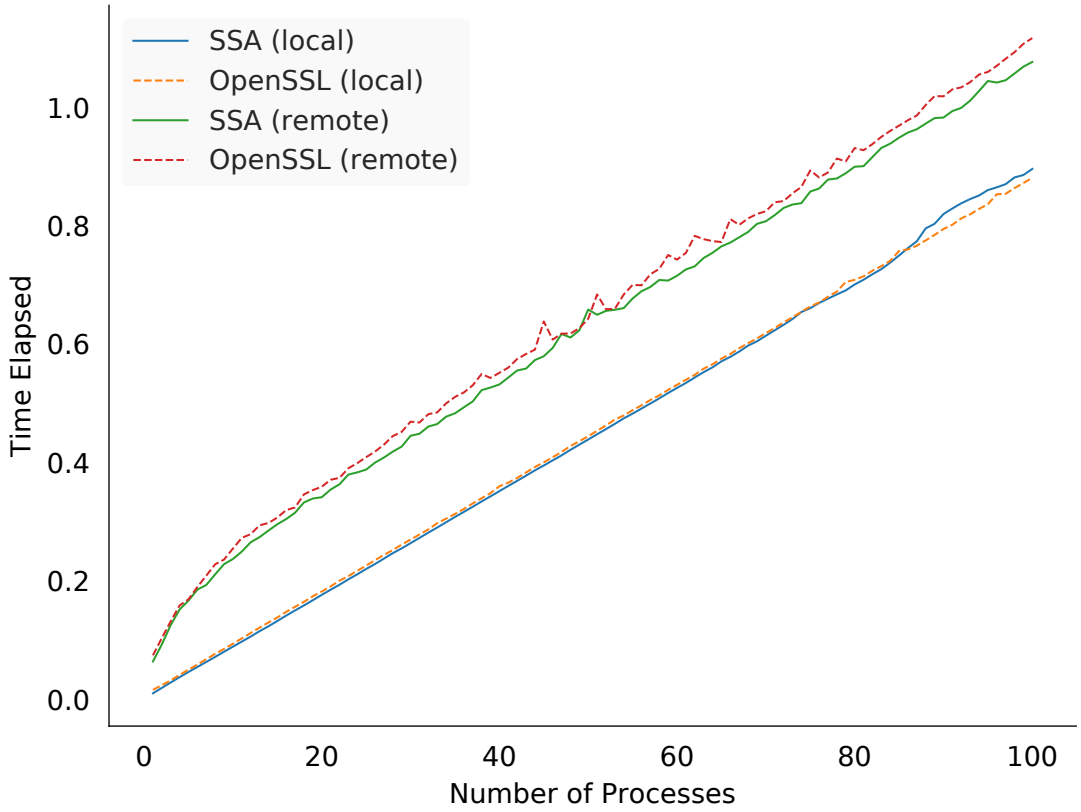



Figure 3.4: Time to transfer 1MB over LAN and WAN via HTTPS for applications using OpenSSL and the SSA, with varying numbers of simultaneous processes.

identical TLS parameters. We created multiple simultaneous instances of these applications and recorded the time required for all of them to receive a remote file over HTTPS, repeating this for increasing numbers of concurrent processes. We show the results of running these tests for 1-100 concurrent processes in Figure 3.4. Each test was run against both local and remote webservers and averaged over ten trials. The machine hosting the applications was a 6-core, hyperthreaded system with 16 GB of RAM, running Fedora 26.

In the local and remote server cases, we find that the SSA and OpenSSL trendlines overlap each other consistently. We use multiple regression to determine the differences between the SSA and OpenSSL timings in both cases. We find no statistically significant difference for local connections ($p = 0.08$) but do find a difference for remote ones ($p = 0.0001$).

For the remote case we find that, on average, the SSA actually improves latency by between 0.1 ms and 0.4 ms per process.

### 3.8 Coercing Existing Applications

In an effort to further support administrators wishing to control how TLS is used on their systems, we explored the ability to dynamically coerce TLS applications using security libraries to use the SSA instead. We focused our efforts on overriding applications that dynamically link with OpenSSL for TLS functionality. Bates et al. [10] found that 94% of popular TLS-using Ubuntu packages are dynamically linked with their security libraries, indicating that handling the dynamic linking case would be a significant benefit.

We supply replacement OpenSSL functions through a shared library for dynamically linked applications to override normal behavior (usable via `LD_PRELOAD`, drop-in library replacement, etc.). This allows us to intercept library function calls and translate them to their related SSA functionality. Under OpenSSL, an application may invoke a variety of functions to control and use TLS. Supplying true replacements for each of these 504 symbols is both cumbersome and unnecessary. Instead, we need only to hook OpenSSL functions which perform operations on file descriptors, and those which provide information necessary for the SSA to perform the TLS operations properly (e.g., setting hostnames, private keys, and certificates). By hooking functions that operate on file descriptors, we isolate an application's socket behavior from the OpenSSL library, allowing the SSA to control network interaction exclusively.

OpenSSL uses an `SSL` structure to maintain all TLS configuration for a given connection, including the certificates, keys, TLS method (server or client), etc., that the application has chosen to associate with the given TLS connection (which is done through other function calls). Our tool obtains the information needed to perform a TLS connection from this `SSL` structure.

When a connection is made on an `SSL`-associated socket, our tool silently closes this socket, creates a replacement SSA TLS socket, and then uses `dup2` to make the new socket use the old file descriptor. Using the associated `SSL` structure, the tool performs the appropriate SSA `setsockopt` calls and then performs a POSIX `connect` on the socket. All socket-using OpenSSL function, such as `SSL_read` and `SSL_write`, are replaced with normal POSIX equivalents (e.g., `recv` and `send`), thereby allowing the SSA to perform encryption and decryption. Since these functions and others have different error code semantics, we also make hooks to change the `SSL_get_error` function to make appropriate OpenSSL errors based on their POSIX counterparts.

During the lifetime of the connection, OpenSSL options set and retrieved by the application are translated to relevant `setsockopt` and `getsockopt` functions, if necessary. For example, the `SSL_get_peer_certificate` function was overridden to use `getsockopt` with a special `TLS_PEER_CERTIFICATE_CHAIN` option to provide applications with X509 certificates to enable custom validation (many applications use this function to validate the hostname of certificates).

Network applications can also create and connect (or accept) a socket *before* associating them with an `SSL` structure. This is typical for applications that use STARTTLS, such as SMTP. To handle this scenario, the tool passes ownership of a connected descriptor to the SSA encryption daemon. The daemon uses this descriptor as its external socket for the brokered TLS connection, and the SSA provides a new TLS socket descriptor to the application for interaction with the daemon.

We abstracted this functionality and added it to our Linux implementation in the kernel component, providing the developer with a `TCP_TLS_UPGRADE` option to upgrade a TCP socket to use TLS via the SSA after it has been connected. This enables applications to use STARTTLS when they find that a remote endpoint supports opportunistic TLS.

In our experimentation with this tool, we successfully forced `wget`, `irssi`, `curl`, and `lighttpd` to use the SSA for TLS dynamically, bringing the TLS behavior of these applications under admin control.

## 3.9 Discussion

Our work is an exploration of how a TLS API could conform to the POSIX socket API. We reflect now on the general benefits of this approach and the specific benefits of our implementation. We also discuss SSA configuration under different deployment scenarios and offer some security considerations.

### 3.9.1 General Benefits

By conforming to the POSIX API, using TLS becomes a matter of simply specifying TLS rather than TCP during socket creation and setting a small number of options through `setsockopt`. All other networking calls (e.g. `bind`, `connect`, `send`, `recv`) remain the same, allowing developers to work with a familiar API. Porting insecure applications to use the SSA takes minutes, and refactoring secure applications to use the SSA instead of OpenSSL takes a few hours and removes thousands of lines of code. This simplified TLS interface allows developers to focus on the application logic that makes their work unique, rather than spending time implementing standard network security with complex APIs.

Because our SSA design moves all TLS functionality to a centralized service, administrators gain the ability to configure TLS behavior on a system-wide level, and tailor settings of individual applications to their specific needs. Default configurations can be maintained and updated by OS vendors, similar to Fedora's CryptoPolicy [58]. For example, administrators can set preferences for or veto specific TLS versions, cipher suites, and extensions, or automatically upgrade applications to TLS 1.3 without developer patches. We have also found that by leveraging dynamic linking, as in Bates et al. [10], applications that currently employ their own TLS usage can be coerced to use the SSA and thereby conform to local

security policies. This can also protect vulnerable applications currently using OpenSSL incorrectly, or using outdated configurations.

### 3.9.2 Implementation Benefits

By implementing the SSA with a kernel module, developers who wish to use it do not have to link with any additional userspace libraries. With small additions to libc headers, applications in C/C++ can use the new constants defined for the `IPPROTO_TLS` protocol. Other languages can be easily modified to use the SSA, as demonstrated with our efforts to add support to Go, Python, and PHP.

Adding TLS to the Linux kernel as an Internet protocol allows the SSA to leverage the existing separation of the system call boundary. Due to this, privilege separation in TLS usage can be naturally achieved. For example, administrators can store private keys in a secure location inaccessible to applications. When applications provide paths to these keys using `setsockopt` (or use them from the SSA configuration), the SSA can read these keys with its elevated privilege. If the application becomes compromised, the key data (and master secret) remain safely outside the address space of the application, inaccessible to malicious parties (`getsockopt` for `TLS_PRIVATE_KEY` is unimplemented). This is similar in spirit to Mavrogiannopoulos et al.'s kernel module that decouples keys from applications [58].

Finally, the loadable nature of the kernel module allows administrators to quickly adopt the SSA and provides an easy avenue for alternative implementations. This is in line with previous Linux kernel security work. The Linux Security Module framework, for example, was created to provide a shared kernel API to access control modules, which allowed administrators to pick the best solution for their needs (e.g., SELinux, AppArmor, Tomoyo Linux, etc.). In a similar fashion, our approach in registering a new TLS protocol allows different kernel modules to hook relevant POSIX socket endpoints for TLS connections and provide unique implementations.

### 3.9.3 Configuration Considerations

The SSA enables administrators and power users to custom-tailor TLS to their local security policies. Enterprise administrators likely have a firm grasp of various policies and their associated implications. However, typical users do not have strong security backgrounds and often rely on their OS vendors for security. With this in mind, Microsoft, RedHat, Canonical, and other vendors could ship their systems with strong default global SSA configurations. These could then be periodically updated according to modern best practices. Some vendors, such as Canonical, already ship application-specific security profiles in addition to global ones [91]. SSA configuration profiles would fit nicely into this model, and also mesh nicely with efforts to centralize security policies, such as Redhat's Fedora CryptoPolicy [57]. Microsoft and Apple could likewise supply global SSA configurations to users of Windows and MacOS, and allow power users to further customize these using the settings UI of these systems. In the mobile space, sometimes operating system updates for devices arrive at rates far less frequent than application updates, as with Android. In such cases, it may be advisable for a vendor, such as Google, to provide SSA configuration (or even the SSA itself) as a system application, where it can be independently updated from the core OS and granted special permissions.

### 3.9.4 Alternative Implementations

POSIX is a set of standards that defines an OS API – the implementation details are left to system designers. Accordingly, our presentation of the SSA with its extensions to the existing POSIX socket standard and related options is separate from the presented implementation. While our implementation leveraged a userspace encryption daemon, other architectures are possible. We outline two of these:

- **Userspace only:** The SSA could be implemented as a userspace library that is either statically or dynamically linked with an application, wrapping the native socket API. Under this model the library could request administrator configuration from default

system locations, to retain administrator control of TLS parameters. While such a system sacrifices the inherent privilege separation of the system call boundary and language portability, it would not require that the OS kernel explicitly support the API.

- **Kernel only:** Alternatively, an implementation could build all TLS functionality directly into the kernel, resulting a pure kernel solution. This idea has been proposed within the Linux community [27] and gained some traction in the form of patches that implement individual cryptographic components. Some performance gains in TLS are also possible in this space. Such an implementation would provide a backend for SSA functionality that required no userspace encryption daemon.

System designers are free to use any of these or other architectures in accordance with their desired practices. The benefit to developers is that they can write code for the same API for all implementations and can pass the burden of TLS complexity to another party.

### 3.9.5 Security Analysis

Our prototype implementation of the SSA centralizes security in the kernel and daemon processes. As such, any vulnerabilities present are a threat to all applications utilizing the SSA. Such risks are part of operating system services in general, as they constitute single points of failure. On the other hand, centralization allows a community to focus on hardening a single design, and security patches to the system affect all SSA-using applications immediately. Given the swift response and incentives OS vendors typically have in responding to CVEs, patches to security systems in the OS will likely be distributed quicker (and more easily) than patches to individual applications. We also note that given the popularity of OpenSSL, it can also behave as a single point of failure, as with the Heartbleed vulnerability.

Another benefit of centralization is that it vastly simplifies the landscape of security problems we face today. At present, thousands of individual applications must each be written to use OpenSSL (or other similar crypto libraries) properly, and experience shows that there

are numerous applications that are at risk due to developer errors. Under the SSA, developer security flaws are likely to be less common, due to the simplicity of invoking the SSA through the POSIX interface and offloading of TLS functionality to the operating system.

Regardless of underlying implementation, the SSA should protect its configuration files from unauthorized edits. Since configuration can affect the security of TLS connections globally, only superusers should be allowed to make modifications. Developers can still bundle an SSA configuration profile for their application, which can be stored in a standard location and assigned appropriate permissions during installation. Many software packages behave similarly already, like Apache webserver packages, which install protected configuration files for editing by administrators.

An existing issue in security is made more apparent by the SSA. The SSA modifies the responsibilities of network security for administrators, operating systems, and developers. As such, it remains in question which party is held accountable when security fails. Implementation bugs can be attributed to the SSA (just like OpenSSL bugs), but vulnerabilities due to improper configurations can be the fault of any of these parties. While we believe that administrators should have the final word over their systems, it is foreseeable that some application developers may want to ensure their own security needs are met, due to legal or other reasons. In such cases, one solution is for developers to ship their applications with a notice that obviates any warranty if the administrator decides to lower TLS security below a given set of thresholds. This issue of misaligned developer and administrator security practices is also present in other security areas, such as running software as a privileged user unnecessarily, making configuration files globally writable, or using sensitive software from accounts with weak login credentials.

## 3.10 Limitations and Future Work

Our exploration has exposed some limitations of our approach, our implementation, and the SSA itself. Each of these has also uncovered potential avenues for additional exploration and expansion of the SSA.

First because we used static analysis of code using `libssl`, we could not determine what code is actually executed during runtime. Performing rigorous symbolic execution or runtime analysis of such a large corpus of packages is outside the scope of our study. As a result we may have overestimated or underestimated the prevalence of use of certain OpenSSL functions. However, static analysis does have the benefit of providing insight into the code developers are writing, which is what led us to find that many developers were expressing TLS options through compilation controls. In addition, we limited our analysis to applications using OpenSSL. The usage of GnuTLS and other libraries may differ in ways that could affect our design recommendations.

Because the SSA targets the POSIX socket API, we believe implementations very similar to ours can be deployed on operating systems that closely adhere to this standard, such as Android and MacOS. Windows also supports this API (with minor deviations), although the mapping between POSIX functions and system calls is not as direct as in the other systems. As such, the kernel module component of our implementation would have to be adapted accordingly.

One limitation of the SSA itself is that it cannot easily support asynchronous callbacks. While we did not find a reason why such a feature was strictly needed for TLS management, it is possible that such a use case may arise. Hypothetically, to support this, `setsockopt` could adopt an option that allowed a function pointer to be passed as the option value. This function could then be invoked by the SSA implementation when its corresponding event was triggered. Under kernel implementations of the SSA, providing arbitrary functions to the kernel to execute seems like a dangerous proposition. In addition, invoking a process

function from the kernel is not a natural task and such behavior seems to be limited to the simplicity of signals and their handlers.

One unexplored path for future work is the suitability of the SSA for network security protocols other than TLS. The QUIC protocol is a prime candidate for experimentation, due to its consolidation of traditionally separate network layers, connection multiplexing, and use of UDP. These features would further test the flexibility of the POSIX socket API for modern security protocols.

## 3.11 Related Work

There is a large body of work that covers the insecurity of applications using security libraries and methods to improve certificate validation in particular, some of which we reference in Section 3.3. Here we outline related work that aims at simplifying and securing TLS libraries, and improving administrator control.

**3.11.0.0.1 Simplified TLS libraries:** `libtlssep` is a simplified userspace library for TLS that uses privilege separation to isolate sensitive keys and other data it uses from the rest of the application, which reduces the payoff for malicious parties exploiting application bugs [6]. This effort resulted in a significant security improvement, but developers still have to learn and interface with the new library, which still requires the addition of hundreds of lines of code for applications. The OpenSSL fork LibreSSL [69] contains `libtls`, a simplified userspace library for TLS that also removes vulnerable protocols such as SSL 3.0. However, nearly a hundred functions are still exported to developers and the library offers no advantage over OpenSSL for administrator control. Secure Network Programming (SNP) [92] is an older security API that predates OpenSSL and SSL/TLS. This API allowed programs to use the GSSAPI to access security services in a simplified way that resembled the Berkeley sockets API (which heavily influenced the POSIX socket API). We further this idea by using, rather than emulating, the POSIX socket API and use it for modern TLS. Collectively, prior

work also largely ignores the suitability of their APIs to languages other than C/C++, which limits their utility to a large amount of developers.

**3.11.0.0.2 Administrator control over TLS:** Fahl et al. [32], MITHYS [15] and two other solutions, TrustBase [66] and CertShim [10], provide administrator and operating system control over TLS certificate validation. Under these systems, an administrator can enforce proper validation by most, if not all, applications on their machines. With the latter three, administrators can even customize certificate validation by employing plugins that strengthen validation (e.g., revocation checks, DANE [39], etc.) As a consequence, these systems remove the burden on developers to implement correct validation. However, these systems fall short of providing administrator control over more than certificate validation, and all but TrustBase only function with applications written in specific languages. In contrast, the SSA provides administrator control of numerous other aspects of TLS (version, ciphers, extensions, sessions, etc.) as well as certificate validation (which can use TrustBase behind the scenes). Apple's App Transport Security [7] (ATS) is a feature of iOS 9+ that mandates that applications use modern TLS standards for their connections. Applications can add explicit exceptions to this as needed, and even disable it entirely. The SSA both enforces administrator preferences and provides a means whereby developers can easily migrate to using modern TLS. While the SSA enables developers to increase security, they are not able to decrease it.

## 3.12 Conclusion

Our work explored TLS library simplification and furthering administrator control through the POSIX socket API. Our analysis of OpenSSL and how applications use it revealed that developers tend to adopt library defaults, make mistakes when specifying custom settings, implement boilerplate functionality that is best implemented by the operating system, and configure TLS usage based on compile-time arguments supplied by administrators. These

findings informed the design of our API, and we find that TLS usage fits well within the confines of the existing POSIX socket API, requiring only the addition of constant values to three functions (`socket`, `getsockopt`, `setsockopt`) to support TLS functionality. In our use of the SSA we find that it is easy to port existing secure applications to the SSA and add TLS support to insecure applications, requiring as little as one line of code. Our prototype implementation demonstrates the API in practice, showing good performance versus OpenSSL. We demonstrate that our implementation can support additional programming languages easily, adding support for three other language implementations with less than twenty lines of code each. We also find that existing applications can be dynamically forced to use the SSA, enabling greater administrator control. Overall, we feel that the POSIX socket API is a natural fit for a TLS API and many avenues are available for future work, especially with alternative implementations.

# Chapter 4

## Modernizing TLS Client Authentication

In this chapter we explore the benefits of placing TLS client authentication in the operating system. Coupled with the latest improvements in TLS version 1.3, the SSA is extended to support strong user authentication that preserves privacy and mitigates phishing attacks. Both server and client applications can use this service, and administrators can control how and which authentications are performed on their machines. The text of this chapter is from the following article, to be submitted for peer review:

> O'Neill, M., Collett, T., Davis B., Coram M., Whitehead J., Perdue T., Seamons, K., & Zappala, D. *A Modern Approach To TLS Client Authentication.*

### 4.1 Abstract

In this work, we explore the viability of a modern approach to TLS client authentication, placing services in the operating system such that applications and users can have a unified and easy way to access authentication functionality. We create a platform for TLS client authentication that is application agnostic, as an alternative to current efforts that are at a high layer and thus are dependent on application support. To facilitate greater user privacy and control, the platform separates TLS authentication devices from the host that establishes the TLS connection. We also discuss new mechanisms to protect sensitive data, isolating these from normal application traffic. We demonstrate the utility and benefits of our approach with a prototype implementation of the platform and an Android authentication application.

We assess usability and gain user feedback through a laboratory user study, and we discuss our findings and the benefits of the authentication platform.

## 4.2 Introduction

Historically TLS client authentication has had difficulty being deployed and adopted. This difficulty is due to technical issues, both for servers and clients [71], as well as usability issues for users [90].

Software developers lack a common platform to employ TLS client authentication, resulting in a fragmented space where browsers and other applications have to individually create their own implementations. For developers less familiar with security, the task of building a custom client authentication system may be prohibitively difficult. Furthermore, coupling TLS client authentication functionality with application functionality can also cause confusion for server configuration and even introduce vulnerabilities [71].

Even if developers can overcome these hurdles, users are faced with additional issues. Users lack an authentication interface with a singular look and feel, which can cause confusion when learning how to use TLS client authentication. Parsovs noted that browsers display their own interfaces to users with differing amounts of options and information, and none offer an accessible "logout" function. In addition, many client applications, and even libraries, like OpenSSL, force users to disclose their private keys directly to an application before TLS client authentication can be used. For example, Firefox requires that users supply a PKCS12 file containing their certificate *and* private key, to be stored by the browser, before users can use the certificate for login purposes. This can be undesirable for users who want to maintain only a single copy of their private key, or in cases where applications may not merit the trust of a user. Together, these problems restrict the security and freedom of users, and make certificates difficult to use.

These issues have made TLS client authentication infeasible for most services and most users today. Our interest is in using modern system approaches whereby TLS client

authentication can be enhanced, so that it can be made more adoptable by developers and users, and be a viable alternative to purely application-layer approaches.

In this paper, we explore the benefits of leveraging the operating system to create a common platform for TLS client authentication. This approach is inspired by recent work that has moved TLS functionality into operating system services. As one example, TrustBase uses kernel-based traffic interception to provide administrator-controlled validation of TLS server certificates, resulting in unified policy control of certificate validation for all applications on a machine [66]. Likewise, the Secure Socket API (SSA) uses operating system services to implement TLS, reducing the developer-facing TLS API to the traditional POSIX socket API [67]. This enables application developers to drastically reduce the amount of code required to deploy TLS and its features, and allows administrators to set policy for TLS settings (such as cipher suites and revocation checking) that are enforced system-wide.

Like this past work, centralization in operating system services has the potential to solve the developer and usability issues that have plagued TLS client authentication for many years. Developer effort to implement applications using client certificates can be drastically simplified, and administrators can set policy for how client certificates are handled. Likewise, users can benefit from a single interface for certificate-based authentication that is identical for all applications. Careful design of the architecture can yield additional benefits, such as separating private keys from applications and separating authentication devices (e.g. a smartphone storing private keys in a secure enclave) from devices used for logging into Internet services (e.g., a laptop).

In our exploration, we make the following contributions:

- An operating system platform for network applications to use for easy deployment of TLS client authentication, for both TLS servers and clients. This platform allows client applications to enable strong user authentication with as little as zero lines of code and allow servers to provide client authentication challenges with as little as one line of code.

- A complete prototype implementation, including a web application and server for an online store, code for the OS platform, and an Android application that demonstrates how to separate the authentication device from the login device. The application allows users to perform certificate enrollment, authentication, session termination, and other authentication tasks. Source code for all of these is provided at `owntrust.org`.
- An analysis of the advances made in TLS client authentication using TLS 1.3 and our authentication platform, with respect to common TLS client authentication issues and those previously noted by Parsovs [71].
- Results and discussion from a user study involving twenty participants that evaluated the usability of the Android application and its connection to our platform. This study also obtained additional insights and opinions from users regarding the use of the application versus passwords, and the benefits and drawbacks of strong authentication.

## 4.3 Background

The TLS handshake protocol is responsible for the establishment of the parameters of a TLS connection and authentication guarantees. In a typical TLS handshake only the server is authenticated to the client, but servers can also request that clients authenticate themselves during this phase as well. In this section we provide an overview of the TLS handshake for server and client authentication.

### 4.3.1 Typical TLS handshake

A typical TLS handshake wherein the client authenticates the server is shown in Figure 4.1. To begin the handshake a client sends a `ClientHello` message to the server, which contains a list of the cipher suites the client supports, with its first preference first. It also contains other data relevant to the security of the connection, such as random bytes from a secure PRNG and various extensions. The server replies to this message with a `ServerHello`, which indicates the cipher suite chosen by the server and its own set of random bytes. This message is

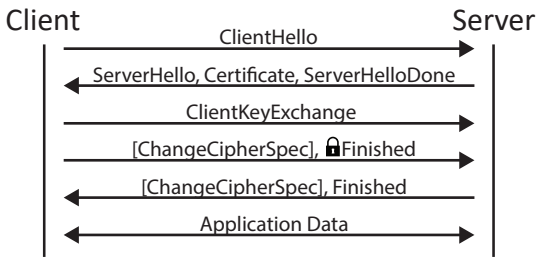



Figure 4.1: Typical TLS handshake using server authentication. The lock icon is shown next to the first message encrypted with the chosen keys and cipher suites All succeeding messages not surrounded by brackets ([]) are also encrypted.

immediately followed by a `Certificate` message, which contains a chain of X.509 certificates for the client to validate, one of which is the server's own certificate containing its public key. The server then indicates it is done with its hello phase with a `ServerHelloDone` message. After properly validating the server's certificate chain, the client sends a `ClientKeyExchange`, which contains a random pre-master secret encrypted with the server's public key.

After this point the client sends a `ChangeCipherSpec` message to the server, which indicates that all successive messages will be encrypted using the negotiated cipher suite and keys. This is followed by a `Finished` message, which is the first message that is encrypted, and contains a hashed transcript of all prior handshake messages, for verification by the server. When the server successfully validates this message, it sends its own `ChangeCipherSpec` and `Finished` messages to the client. Once both parties have verified their peer's `Finished` messages, the handshake is complete and application data can then be sent and received, under the protection of the negotiated cipher suite.

### 4.3.2 Handshake with Client Authentication

To perform client authentication, the handshake process described is augmented slightly, as shown in Figure 4.2. The server is responsible for requesting client authentication, and does so by sending a `CertificateRequest` message to the client, following its `Certificate` message. In this case, the client will send its own `Certificate` message containing its own chain of X.509 certificates to the server, immediately after receiving a `ServerHelloDone`. Note that

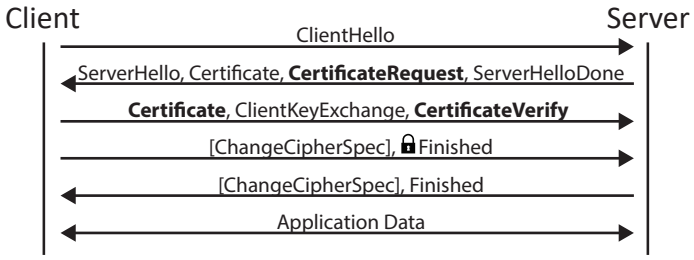



Figure 4.2: TLS handshake using mutual authentication. The lock icon is shown next to the first encrypted message. All successive messages not surrounded by brackets ([]) are also encrypted.

the `Certificate` message is sent by the client in the clear, before encryption begins. To prove possession of the private key for its certificate, the client also sends a `CertificateVerify` message following the `ClientKeyExchange` message. Within this message is a signature over all previously sent and received handshake messages, created using the client's private key. This binds the authentication to the current TLS handshake. If the `Certificate` and `CertificateVerify` message provided by the client are found to be trusted and valid by the server, respectively, then the client is successfully authenticated.

Traditionally, servers request client authentication by first triggering a TLS renegotiation. Under a renegotiation, a new TLS handshake is performed, but its messages are protected by the encryption of the existing TLS session. This allows servers to selectively require client authentication based on requests to specific protected resources. It also provides users with more privacy, since client certificates are not sent encrypted during the initial TLS handshake.

### 4.3.3 TLS 1.3 Advances

TLS 1.3, recently standardized, modifies the handshake protocol, simplifying the number of required messages and increasing security. Figure 4.3 and Figure 4.4 depict the TLS 1.3 handshake without and with client authentication, respectively. While a full discussion of the changes under version 1.3 are outside the scope of our discussion, two important differences from version 1.2 are of note. First, TLS 1.3 encrypts messages as soon as it can.

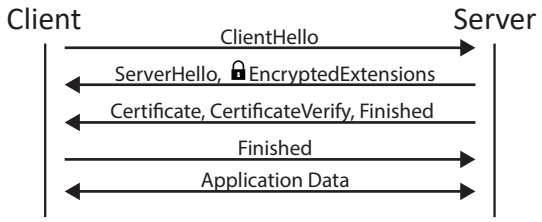



Figure 4.3: TLS 1.3 handshake. The lock icon is shown next to the first encrypted message. All successive messages are also encrypted.

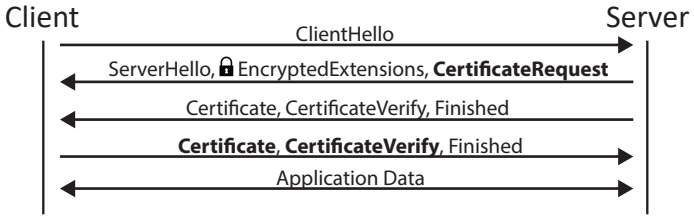



Figure 4.4: TLS 1.3 handshake using mutual authentication. The lock icon is shown next to the first encrypted message. All successive messages are also encrypted.

Everything following the `ServerHello` message is encrypted using the negotiated cipher suite. As a consequence, the `Certificate` message from the client is protected from network eavesdroppers, obviating the need to perform a renegotiation to protect this message. Second, the server itself is responsible for sending a `CertificateVerify` message, allowing the client to obtain proof that the server owns the private key corresponding to its certificate before the client is required to send its own certificate.

In addition to these advances, TLS 1.3 also supports "post-handshake authentication" (PHA). Using this mechanism, at any point during a TLS connection, the server may request that the client authenticate to the server. After receiving this request, the client replies with `Certificate`, `CertificateVerify`, and `Finished` messages. If the client does not wish to authenticate, it can send an empty `Certificate` message followed by a `Finished` message. This functionality allows servers to selectively require authentication depending on the resources requested by the client in a more natural way (no re-handshaking required).

```
/* Request a TLS socket (instead of TCP) */
fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TLS);

/* Specify key and cert to use for auth */
setsockopt(fd, IPPROTO_TLS, TLS_PRIVATE_KEY, KEY_FILE, sizeof(KEY_FILE));
setsockopt(fd, IPPROTO_TLS, TLS_CERTIFICATE_CHAIN, CERT_FILE, sizeof(CERT_FILE));

/* TLS Handshake */
connect(fd, &addr, sizeof(addr));

/* Communicate with server */
send(fd, ...);
...
recv(fd, ...);
...
shutdown(fd, SHUT_RDWR);
```

Figure 4.5: Example SSA-based TLS client using client authentication. Error checking and other code is omitted for brevity.

### 4.3.4 Secure Socket API

The SSA is an API that allows developers of network applications to easily employ TLS. When using the SSA, developers actually use the default POSIX socket API composed of familiar functions like `socket`, `connect`, `bind`, and `listen`. Rather than specifying TCP or UDP, developers can select a protocol of `IPPROTO_TLS` when creating a socket, and subsequent read and write operations on that socket use TLS behind the scenes. This drastically reduces the amount of code required to use TLS and centralizes TLS as an operating system service.

Developers can customize TLS behavior using the `getsockopt` and `setsockopt` functions. As an example of SSA use, Figure 4.5 depicts a TLS client using the SSA that also employs client authentication. In this case, the `TLS_CERTIFICATE_CHAIN` and `TLS_PRIVATE_KEY` options were used to specify paths to the certificate chain and key to use for this connection, respectively. Developers also have access to a variety of other options, including the ability to terminate sessions, and request the certificate chain of the remote peer for validation. We note that in the latter case, the SSA itself performs its own validity checks on behalf of client applications in accordance with local system policy.

## 4.4 Problems

TLS client authentication is currently faced with several problems that limit its use and adoption. These fall loosely into three categories: those faced by servers and administrators, by clients and users, and by public key infrastructure.

### 4.4.1 Servers and Administration

Parsovs [71] cataloged a variety of issues with the use of TLS client authentication. His focus was primarily issues from a server administration perspective, but also included some treatment of user issues as well. These particular insights are of note as they were derived from the experiences of service providers in Estonia, a country which provides digital certificates, via smart cards, to citizens when they reach fifteen years of age. Parsovs noted numerous developer mistakes in Apache's `mod_ssl` that made server administration difficult when deploying TLS client authentication. Some of these related to configurations that created difficulties for administrators. These included confusing directives for access control, a lack of audit trail, and inflexible certificate validation – including the inability to perform custom validation or enhance it with revocation checking. Other issues were related to various vulnerabilities, such as denial of service against an RSA implementation and an attack that allowed attackers to determine the server's trust settings remotely.

One particular egregious issue was the inability for server administrators to properly determine when clients had sent their `CertificateVerify` messages, which in turn leads to a vulnerability that allowed attackers to have an arbitrary amount of time to craft forged messages to impersonate users. This last issue arises in part due to a mixing of the security and application layers. While `mod_ssl` expects an identity attestation message from the client, and should terminate the connection after a given expiration, the webserver portion of Apache attempts to keep connections alive as long as data is being sent. This confusion of application data with security data should not occur, but server developers are currently forced into a situation wherein they must do this when deploying TLS client authentication.

### 4.4.2 Clients and Users

Parsovs also mentioned a variety of other issues with browser implementations of TLS client authentication, and client privacy issues. When a server requests TLS client authentication, browsers each display different dialogues to users, each with a differing amount of detail and different semantics. Aside from the UI fragmentation, browsers do not offer an easy way for clients to dictate how long to use a certificate for a remote host, or provide an option to terminate the session and cease authentication. In many cases, sessions must be manually terminated by the user, through process termination or cache clearing. These difficulties and UI inconsistencies make using TLS client authentication difficult. As with the server case, client application developers are forced to reimplement TLS client authentication functionality and UI design for each application, unable to rely on a common platform for these services. This problem is even more important outside of the browser space, where the lack of technical knowledge required to implement all of this functionality may preclude its inclusion altogether.

Many issues surrounding TLS client authentication pose threats to the security and privacy of users themselves. It is well-known that TLS 1.2 and below transfer the user's certificate in the clear during the initial handshake, and this precluded the use of TLS client certificates by some technologies [8]. TLS 1.3 has resolved this issue, but users are not usually made aware of the version of TLS that their applications use, nor the associated privacy implications. Another, less noted set of user issues surround the interaction between the application and a user's credentials. Currently, browsers and other applications often require users to disclose their private key to the application, to allow the application to generate `CertificateVerify` messages. For example, Chrome requires users to insert a PKCS #12 file containing their certificate and private key into the browser for use. This is undesirable for private key best practice, and impossible when using hardware tokens that prohibit private key export. In addition to the disclosure of the private key, users have to submit their certificate information to applications as well. While this occurs out of necessity, due

to the fact that TLS itself transmits the certificate to the remote server, currently no effort is made to remove the certificate from application visibility. This is relevant for anonymity, wherein a user may wish to authenticate to a remote host securely, but may not trust the client application with identification information, such as a certificate.

### 4.4.3 Public Key Infrastructure

Usability issues have long-plagued consumer public key cryptography security solutions, and TLS client authentication is no different in this regard. Parsovs correctly noted that a PKI infrastructure is not strictly required for the deployment of client authentication, and that servers typically do not need to have any proof of identity when a new user is registering for an account. In this sense, client certificates can be self-signed, and users do not need to trouble themselves by obtaining a certificate from a trusted authority. However, there are some lingering issues that are not addressed by a PKI-less system. Without a certificate issued from a CA, which is often the case outside of enterprise environments, the tasks of enrollment, renewal, and revocation are more complicated. Modern browsers can support enrollment in a PKI-less way using `CertEnroll` (Edge) and the HTML5 `keygen` element. However, our focus in this work is generic TLS and we do not want to limit solutions to only the browser space. In the password authentication realm, revocation and renewal are easily addressed on the web, through the use of password reset links, which typically email a special token to the user which can be redeemed to reset a password. In these cases, access to an email account is considered sufficient proof of identity, although some systems require users to provide further information, through answering a series of questions about their account use or "security questions". Unfortunately, these mechanisms have no analog in the PKI-less client certificate space, nor is there an automated system or platform on which developers can rely to assist their users.

## 4.5 Threat Model

In this work we adopt a threat model wherein users are attempting to use TLS client authentication on a computer system (host computer), in enterprise, home, and even public settings, such as a public library computer. We place minimal trust in applications installed on the host computer, and only trust the host computer's operating system with the minimal set of user-supplied information required to broker a mutually-authenticated TLS connection between the user and a remote service. Users operate authentication devices, which we call authenticators (mobile phone apps, smartcards, hardware tokens, etc.) to supply identity data to the host computer, but otherwise use a typical mouse and keyboard (or other traditional input). Malicious or vulnerable software may be present on the host computer, which may be actively attempting to steal user credential information, or inadvertently allow attackers to obtain such data. Information potentially leaked to attackers includes both private keys and certificates. However, we do not consider any information leaked through application layer data or behaviors. For example, a user posting her name onto an online forum via HTTPS is outside the scope of this model. Given their absolute privilege and unrestricted visibility, in our model we consider the kernel of the operating system and privileged processes to be trusted. However, we refrain from trusting the operating system with more than the minimal set of information it requires to provide its services.

While we primarily target the situation in which an authenticator is a separate physical device from the host computer, we also consider situations in which these may be the same device. For example, this is the case when an authenticator is implemented as a mobile application and a user uses it to authenticate to a service from the phone's browser. In such cases, the authenticator is considered a local, software-based authenticator. We assume that local authenticators use modern hardware support (e.g., an HSM) to isolate private key and other identity data from arbitrary access by other applications or the system. Furthermore, in this case, we do not consider phishing techniques from local malware that hijack the user authenticator interface to trick a user into logging into an imposter service.

## 4.6 Design Goals

In this work we seek to design a platform to modernize TLS client authentication, helping developers to easily deploy it, administrators to manage it, and users to leverage it. Specifically, our goals are as follows.

- Provide a platform for TLS client authentication. Application developers should be able to rely on simple function calls to deploy and use TLS client authentication in their systems. This goal includes both server and client-side components, with an emphasis on the latter.
- Provide flexible and easy configuration. The platform should be secure by default, preventing users from authenticating when TLS is being used insecurely, such as when TLS 1.2 servers request authentication on the first handshake. Likewise, server administrators should be able to easily configure applications based on the platform, indicating how certificate checks are performed and enabling advanced features, such as CRL checks, when desired.
- Provide a generic protocol for interaction with authentication devices. We target local authentication devices that are present on the same device used to perform the TLS connection, as well as external ones connected through some medium to a host system. To reduce the fragmentation of UIs for TLS client authentication, and assist in user choice, the platform should provide a generic protocol to allow authentication devices to provide and obtain relevant security messages to and from a host system to influence a given TLS connection. Authentication devices can be software or hardware-based, and connect to the platform via software protocols or physical protocols, such as BlueTooth or WiFi. In addition to providing login functionality, authentication devices should be able to provide logout functionality by forcing the destruction of any active TLS sessions or tickets on the platform.

- Provide isolation and control of sensitive data. Applications should be allowed to communicate over a secured channel, but not obtain direct access to information that can impersonate or identify a user, unless this information is transmitted by the user or server through the application layer. This includes isolating both the certificate and private key from client applications, and allowing private keys to remain within the domain of the authentication device hosting it.
- Provide privacy from network eavesdroppers and client applications. We seek to isolate the identifying information in certificates from applications and network eavesdroppers. We can support the latter of these by default when using TLS 1.3 and secure renegotiation with TLS 1.2, preventing the disclosure of certificates in plaintext during a TLS connection, which has historically been an issue with TLS [88].
- Provide usability. The platform should be usable, both by developers of client and server components as well as by users themselves. Users should be able to connect their devices to the platform, and perform authentication tasks easily. While this is likely to be an on-going effort as new server applications and user authentication devices emerge, we seek to provide a usable prototype assessed in a laboratory setting.
- Remain application agnostic. While browsers and the web have begun to adopt technologies for strong client authentication, such as with WebAuthn, we seek to provide support for TLS client certificate authentication in a manner agnostic to the application or protocol being used on top of TLS. We also wish to refrain from modifying TLS in any way, to maximize ease of deployment and integration with existing TLS implementations.

## 4.7 A Modern TLS Client Authentication Platform

In this section we describe a new platform for TLS client authentication, based on the aforementioned goals. Our TLS client authentication system uses the Secure Socket API

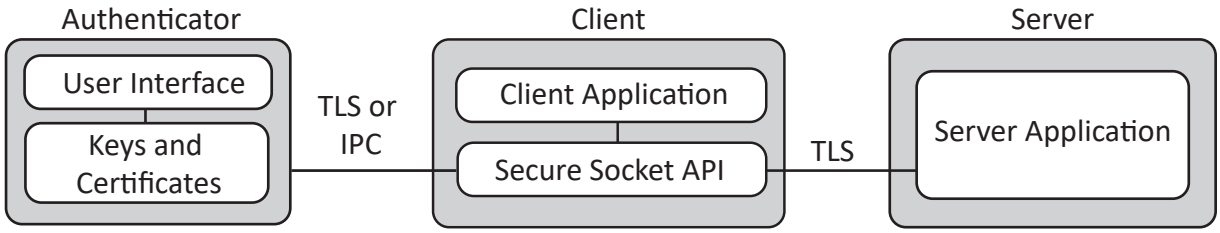



Figure 4.6: Overview of components. Note that the authenticator can access the SSA on the TLS client system via TLS over a communication medium like WiFi, or locally through an inter-process communication (IPC) mechanism.

(SSA), which is an operating system service that provides a simple TLS interface to applications via the POSIX socket API [67]. The use of this API allows us to build upon an existing service, and leverage its isolation of TLS functionality from applications, which assists us in our privacy and security goals.

Figure 4.6 depicts a high-level view of the components of our design. A TLS-enabled server running an arbitrary application layer protocol (i.e., not necessarily HTTP) is accessed by a client application. This connection is facilitated and managed by the SSA, which transfers application layer data to and from the client application. When client authentication is requested by the server — and the client application does not directly specify its own key and certificate — the SSA requests authentication from a connected authenticator. The authenticator maintains its own set of keys and certificates on behalf of a user (or another application), providing the certificate and signatures as needed when authorized by the user. The private keys associated with certificates never leave the authenticator, which can be an external device such as a mobile phone.

### 4.7.1 Indirect Authentication

By default, the SSA provides the ability for client applications to specify the certificate and private key to use during TLS client authentication using `setsockopt` with the `TLS_CERTIFICATE_CHAIN` and `TLS_PRIVATE_KEY` options. However, this does not work for client authentication cases where the certificate to be used is not known before client au-

thentication is requested (during the handshake or using post-handshake authentication). OpenSSL provides developers with the ability to specify a callback for this case, allowing applications such as web browsers to prompt the user for credentials when it encounters a server requesting client authentication.

However, there does not exist any functionality that allows the private key to remain isolated from OpenSSL, forcing users to divulge their keys to the application. This limitation sheds light on an interesting duality with TLS client authentication: direct and indirect cases. In a direct authentication scenario, the client is the actual application performing the TLS connection. This might be used by the developer to simply identify an official client, rather than a user, and thus the private key is already under the control of the application. However, in an indirect case the entity being authenticated is not the application itself, but a party using the application. In this scenario the application may not merit sufficient trust from the authenticated party to have access to the private key, or the authenticated party may wish to maintain only a single copy of the private key (e.g., in an external fob or other hardware token). The client authentication support in the SSA and OpenSSL are well-suited for the direct case, but neither support the indirect case.

We provide support for indirect client authentication through an operating system service. An "authentication daemon" provides this service to applications on a host system. When TLS applications receive requests for client authentication, relevant data is forwarded to the authentication daemon. The daemon, in turn, communicates with local or external authentication devices using a generic protocol, to obtain the information needed to craft a proper response to the request and forward it to the server. During this process, private key data is maintained within the authentication device.

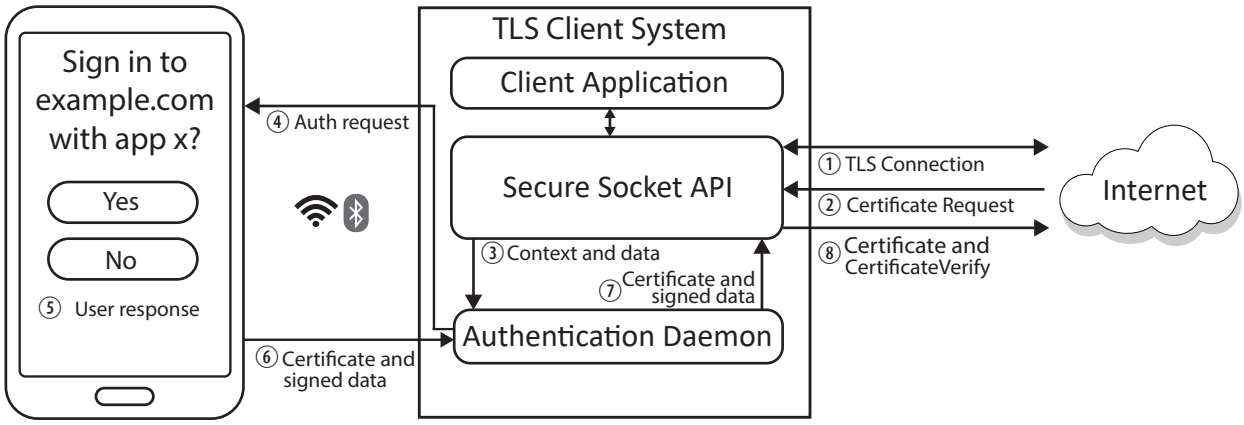



Figure 4.7: Example flow of TLS client authentication using a mobile phone as the authenticator

### 4.7.2 Authentication

An overview of the flow of information in our system is depicted in Figure 4.7. The steps indicated cover the sequence of events that occurs when TLS client authentication is performed. We describe each of these steps in greater detail:

1. A TLS connection is established between a client application and a server, using the SSA. This also includes connections wherein the handshake has not fully completed.
2. The server requests authentication from the client. This can occur during an initial handshake (more secure in TLS 1.3), during post-handshake authentication (TLS 1.3 only), or as the result of a renegotiation (TLS 1.2 and below).
3. The SSA connects to an authentication daemon, which is a local process that facilitates communication between SSA worker processes and a user's authenticator. The SSA provides the authentication daemon with the data to be signed by the authenticator, as well as contextual information about the client application and the remote server.
4. The authentication daemon queries a connected authenticator with the information provided to it by the SSA.
5. The user receives a notification on the authenticator and is presented with a dialogue that informs them of the type of authentication requested, the remote host and application

making the request, and the information that will be disclosed to the remote host upon acceptance.

6. Following approval from the user, the authenticator signs the provided data using a private key, and forwards this signature and the corresponding certificate to the authentication daemon. If a user opts to ignore or deny the authentication request, a failure response is sent to the authentication daemon.
7. The SSA receives the response from the authentication daemon and crafts it into the TLS Certificate and CertificateVerify messages. Note that only an empty Certificate message is crafted in case of authentication denial by the user.
8. The crafted TLS messages are forwarded to the remote server, completing the authentication request.

### 4.7.3 Registration

Account registration links a user's account with a particular certificate or public key. TLS itself does not specify how certificates should be issued or a PKI model for trusting them. In our design, we aim to provide a flexible framework to allow multiple types of account registration, and provide the authenticating user with fine-grained visibility into privacy. When requesting client authentication using a `CertificateRequest` message, the server provides a list of Distinguished Names (DNs) of certificate authorities from which it expects a client certificate to be issued. Under TLS 1.3 the server can use the Object Identifier Filters (OID Filters) extension to request certificates having specific values for given attributes. An empty DN list indicates that the server accepts any certificate, including self-signed ones. Under our system, the list of DNs and other certificate filters are provided to the authentication device by the authentication daemon. Depending on these data, the authentication device can choose one of three options:

1. Generate a self-signed certificate (PKI-less). If the list of DNs is empty, then under TLS semantics the device is allowed to present any certificate to the server. In the event that the authentication device does not currently have a certificate with all the fields required by the server, it can generate its own. Using this option, a device can maintain a distinct keypair (and certificate attributes too, if desired) for each account it manages.

2. Send a Certificate Signing Request (CSR) to the server. If the server has indicated its own hostname within the DN list, the authentication device can opt to obtain a signed cert directly from the server. It does this by connecting to the server on a designated port directly, or tunneling this connection through the authentication daemon if the device does not have network access of its own. The connection port is specified with a custom OID filter value that the authentication daemon reads and forwards to the authentication device. The server then provides a list of identification attributes to the authentication device, marking vital attributes (if any) as mandatory. The device then generates a certificate signing request with all of the attributes the user wishes to include (which may or may not be more than the minimum required set), and sends it to the server. In response, the server provides a signed certificate for use by the device during TLS client authentication. This allows servers to perform validation on the attribute values submitted by the client before the certificate is used in a TLS handshake.

3. Utilize a third-party certificate signed by one of the DNs. If the server has specified some other CA in its DN list other than itself, and the authentication device already has a certificate from that CA, then it can re-use the corresponding certificate for the TLS connection. This is similar to single-sign on (SSO) technologies, which allow a user to log in to a website using credentials from another, usually widely-used one.

When selecting one of these options, the user is informed of privacy implications in two ways. First, the user can be informed that using an existing certificate affords less

privacy than generating a self-signed one or a new one signed by the server. Second, the attributes stored within the certificate to be generated or selected can be provided to the user, allowing them to see clearly what information they will be disclosing to the server upon account registration. In cases where a new certificate is to be generated, the user may also select which attributes to include (self-signed case) or opt-in to providing more information than required by the server (CSR case).

### 4.7.4 Renewal and Recovery

Providing servers with a method of requesting specific certificate attributes during registration creates an easy avenue for PKI features like revocation and renewal. Servers can request the inclusion of these attributes in certificates directly, using their own signing daemon described in the previous section, or indirectly, by requesting that the client use a certificate from another authority. The attributes requested might include a phone number, email address, citizen identification number, or some other verifiable attribute. By challenging ownership of these attributes, a server can conditionally revoke, renew, and issue certificates. For example, requiring that a user prove ownership of an email address before issuing (or accepting) a certificate with that email address as a certificate attribute allows servers to provide key revocation and certificate renewal through a user's email account. This mechanism enables behavior similar to the de facto standard of modern account recovery: password reset via email.

### 4.7.5 Securing Local Connections

In line with our goal to create a generic protocol for TLS authentication devices, we allow the physical connection between devices and the authentication daemon to be wired or wireless. While some authentication devices may use USB, others may use Bluetooth, NFC, or even 802.11 (WiFi). As such, it is important that this connection be protected from man-in-the-middle attacks wherein attackers impersonate a client machine to the authentication

device. TLS is used to secure this communication channel. When an authentication device becomes aware of a new pairing between it and a client machine, it authenticates the machine by verifying its public key out of band. To support this, when a user's phone, acting as an authentication device, connects to a desktop computer for the first time, the computer displays a QR code on its monitor representing a hash of its public key. The user verifies this key by scanning the QR code with the phone, and the underlying authentication software verifies that the key received in the TLS handshake matches the one advertised. Subsequent connections of the authentication device to the same host do not require this step, unless the host changes its key (e.g., due to periodic key rotation).

## 4.8 Implementation Details

In this section we describe the implementation of our platform as well as a prototype authentication device, which we created as an Android application for smartphones. Both the code for the platform and the application are available at `https://owntrust.org`.

### 4.8.1 Serverside Support

While the client components we provide can be leveraged by any TLS-enabled server, we offer a server-side component to make deployment of TLS client authentication easier for new server programs. Using the SSA, and extending it slightly, allows us to elegantly solve many of the prior problems in TLS client authentication.

Figure 4.8 depicts a simple source code example of a server requesting TLS client authentication using our SSA additions. Our first addition to the SSA is the new socket option, `TLS_REQUEST_PEER_AUTH`, which triggers a renegotiation with a certificate request under TLS 1.2, and triggers post-handshake authentication under TLS 1.3. The parameter to this option is an optional set of object identifier (OID) filters that allow the developer to specify a set of attribute value pairs a client certificate should have. A NULL set indicates that any attributes are acceptable, and in such cases the client certificate will be validated according to SSA

```
/* Set up trust store using developer SSA trust store option (optional) */
setsockopt(client_s, IPPROTO_TLS, TLS_TRUSTED_PEER_CERTIFICATES, CA_FILE, sizeof(CA_FILE));
...
/* Request client authentication */
setsockopt(client_s, IPPROTO_TLS, TLS_REQUEST_PEER_AUTH, &oid_filters, sizeof(oid_filters));
...
/* I/O suspended on socket until client response/timeout */
...
/* Obtain client information (optional) */
getsockopt(client_s, IPPROTO_TLS, TLS_PEER_CERTIFICATE_CHAIN, &chain, chain_len);
```

Figure 4.8: Example SSA server request for TLS client authentication. Error checking and other code is omitted for brevity.

policy only, which includes proper signatures back to trusted authorities. Under the SSA, set of trusted authorities can be set by the developer using the `TLS_TRUSTED_PEER_CERTIFICATES` option (seen on the first line in Figure 4.8), or by the administrator.

When client authentication is requested by a server using the SSA, data will not be sent or received until the client has responded to the request, indicated by the reception of the client's (possibly empty) Certificate message. If the underlying socket is blocking, I/O operations such as `recv` and `send` will block during this time. Otherwise, I/O operations will return immediately and set a busy error (`EAGAIN/EWOULDBLOCK`). After the client responds to the request or times out, event-driven I/O constructs such as `select` and `epoll` will be properly notified.

After the client response, developers can perform more advanced or custom client certificate validation, if desired. The SSA allows direct access to the client's chain of certificates using `getsockopt` with the `TLS_PEER_CERTIFICATE_CHAIN` option. An added `TLS_PEER_IDENTITY` socket option can also be used to obtain simplified information about the authenticated client, such as a Common Name value.

The changes described to the SSA allow server applications to request TLS client authentication with as little as one line of code. With an additional line, information about the authenticated client is obtained, including any information stored in the client certificate. In this fashion, servers using the SSA can implement a strong authentication system very easily, lowering the bar to entry for new systems. We note, however, that the SSA does not

provide a mechanism to store and retrieve data associated with an authenticated client, so the server must employ some type of database for this, using the peer identity information as a key.

We implemented TLS client authentication in a custom webserver, using our server-side SSA extensions. When sensitive HTTP endpoints are requested by clients, the server issues a certificate request using the new socket option before it parses the request body or sends a response. The time of authentication is stored by the SSA, and accessible to the application using the data returned by a `getsockopt` call with `TLS_PEER_IDENTITY`. This allows the server to calculate authentication expirations according to its own policies, and, if desired, expire the underlying TLS session using the `TLS_SESSION_TTL` socket option. The webserver exports the peer identity information to CGI processes, such as PHP, allowing web developers to bind session information to these data, including (or in replacement of) HTTP cookies. This also allows the web application to manage its own account database.

Optionally, server administrators can configure their implementation to operate a simple certificate-issuing service upon boot. This service will provide a port number within the `CertificateRequest` message indicating to clients where to send a CSR to obtain a signed certificate. This server implements the protocol described in Section 4.7.3.

### 4.8.2 Authentication Daemon

Building a central authentication service requires handling cases in which multiple applications request service concurrently. For example, a web browser may visit a website requiring authentication at the same time a cloud storage application wishes to synchronize with its servers. We serialize these requests, allowing users to authorize authentication procedures one at a time. To implement this, we extend the SSA to provide an "authentication daemon". The authentication daemon is responsible for coordinating efforts between the authentication devices and the SSA worker processes managing TLS connections for applications. The

authentication daemon allows only a single authentication device for each user on a machine to be connected at a time.

To support indirect authentication, we modified OpenSSL (the security library upon which our SSA implementation depends) to support a new function, `SSL_set_client_auth_cb`, which allows an application to register a callback function to be invoked when a remote TLS server has requested client authentication. Unlike the similar, existing function, `SSL_CTX_set_client_cert_cb`, this new callback does not require the application to provide OpenSSL with the private key associated with the chosen certificate. Instead, when invoked, the callback provides the application with the parameters necessary to choose and use a private key to sign an array of bytes. The byte array is a transcript of all of the handshake messages sent thus far in the connection, so that a digital signature on the array can be used in a `CertificateVerify` TLS message. The SSA uses this feature behind the scenes to provide native, automatic client authentication support to applications. That is, the SSA enables indirect authentication by default, requiring zero lines of code from application developers. If an application has not set the `TLS_CERTIFICATE_CHAIN` and `TLS_PRIVATE_KEY` options, then any request for authentication from a server will trigger the SSA to request services from the authentication daemon. If these options are set by the client application, direct authentication (i.e. authentication of the application itself) is performed, using the associated values.

The messages sent between the authentication device and daemon can be used with a variety of communication mediums, such as Bluetooth Low Energy, USB, and 802.11 (Wi-Fi). The first two have been used by other protocols that secure the connection between an authenticator and computer, such as CTAP. In this work we explore the use of authentication over Wi-Fi. The use of Wi-Fi for this purpose has some benefits with respect to other protocols. For instance, it is common that home computing devices are all connected to the same Wi-Fi enabled router, either through wired or wireless connections. This means that these devices are accessible to users even if their host machines do not have specialized hardware, such

as a wireless antenna. For example, a wired desktop computer can indirectly communicate via WiFi devices through its local router. Furthermore, users are already familiar with the process of connecting their handheld devices to their home networks and others to which they come in contact, and the connections are often remembered and automatically re-established. The primary drawback is that a single Wi-Fi network is shared by many devices, unlike Bluetooth pairing between two devices. This means that multiple hosts can be advertising authentication daemons on a single network, requiring the authentication device to distinguish between them.

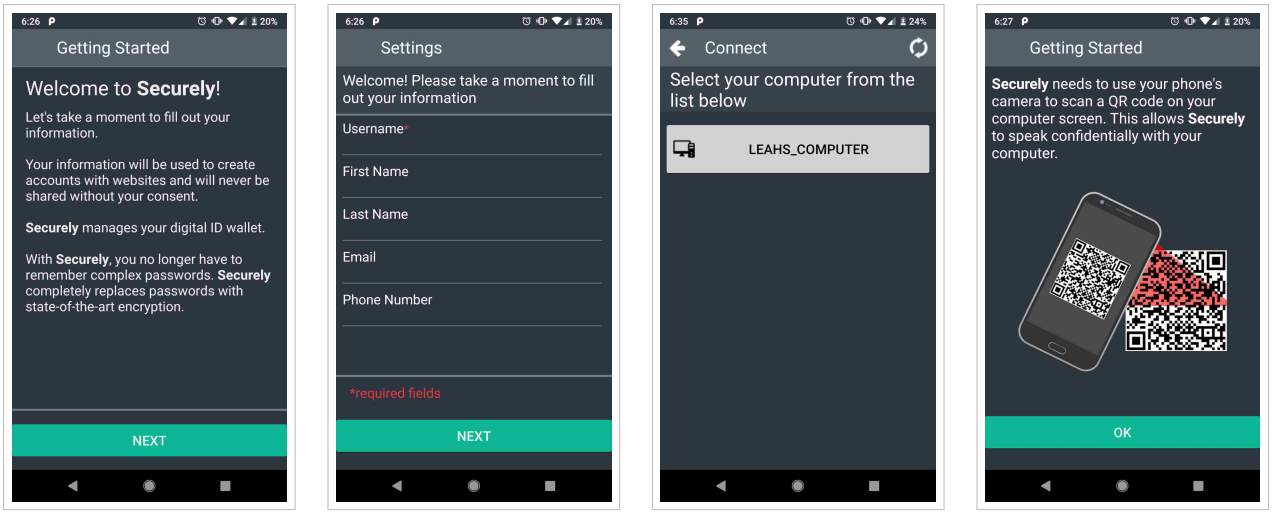
The authentication daemon leverages multicast DNS to advertise its services to potential authentication devices. The DNS messages are sent at regular intervals to the local subnet, advertising the name, public key, IP address, and port number of the authentication daemon. Using this information, authentication devices connected to a Wi-Fi network can connect to the daemon of the user's choice, and remember it based on advertised keys and names.

The authentication daemon provides a generic protocol for facilitating various authentication requests from SSA worker processes and connected authentication devices. The following subsections outline these and their purpose.

#### 4.8.2.1 Connect and Disconnect Request

When an authentication device connects to the daemon through a wireless connection, a TLS connection is established between them. After connection establishment, the device sends a connect request to the authentication daemon. The host may reply with a failure message if another authentication device is already connected, or no users are currently logged on to the machine.

The connection request can include a parameter indicating that the authentication daemon should present its public key for manual validation. In this case, the daemon displays a QR code representation of the key on the local screen, allowing device users to establish




(a) Welcome screen (b) Default info (c) Host selection (d) QR scan notice

Figure 4.9: Screenshots from the Securely authentication app

the validity of the key presented in the connection. In subsequent connections to the same host, the authentication device can opt to forego this validation step, assuming that it has verified that the key from the current connection is the same as a previously validated one.

When an authentication device terminates the TLS connection between it and the authentication daemon, due to manual termination or connection timeout, the authentication daemon considers this a disconnect and no longer attempts to forward authentication requests to the device.

#### 4.8.2.2 Authentication Requests

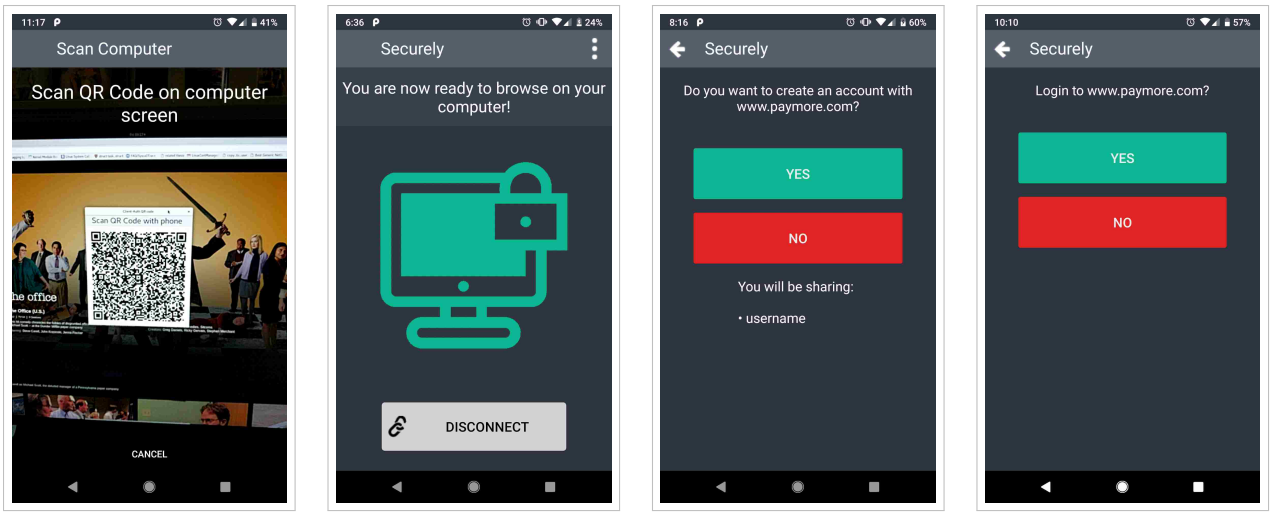
When the authentication daemon receives a request from an SSA worker process, the daemon performs the following actions:

1. If an authentication device has been connected to the daemon, the daemon issues a request to that device. This request includes the hostname of the remote server requesting authentication, the name of the client application, the data to be signed (handshake transcript), the hash algorithm to be used, the signature scheme to be used, and the certificate filters sent in the `CertificateRequest`.

If no authentication device has been connected to the daemon, the daemon displays a notice to the user of the application needing authentication service. This notice is graphical if the current user is running a GUI, or text-based for terminal users. In either case, the notice informs the user that he should connect his authentication device to complete login.

2. If a user authorizes the login, the authentication device responds to the daemon by transmitting a chosen X.509 certificate and a signature over the handshake transcript. If the login request is denied by the user, the device notifies the daemon of the refusal.
3. The authentication daemon forwards the relevant login data to an SSA worker process, which in turn uses that information to complete the TLS handshake or post-handshake authentication response.

An authentication request is analogous to a webform prompting a user for a username and password. That is, the information requested from the user is made known, as well as the identity of the requesting host, but nothing secret is communicated. In response to this request, the authentication device discloses sensitive information, just like a user who logs into a website, and it is therefore necessary for the user to authenticate the authentication daemon with the QR code, as discussed previously. However, an attacker may attempt to register his own device before the user, resulting in a type of denial of service attack. Furthermore, the attacker operating in this fashion may determine the identity of the host to which a user is attempting to authenticate, resulting in some privacy loss. Given that the attacker would require a device connected to the local machine or WiFi network, this may not be a serious concern. However, one mitigation technique is for the daemon to remember a public key used by a device upon first connection and require explicit user approval when a device with an unfamiliar public key attempts to connect to the daemon.




(a) QR code scanning (b) Host ready (c) Registration (d) Login

Figure 4.10: Screenshots from the Securely authentication app

#### 4.8.2.3 Logout Request

To facilitate generic user logout functionality, the authentication daemon listens for logout requests from authentication devices. These requests specify an application and host pair for which the user would like to terminate his or her session. This can be sent in anticipation of changing accounts for an application, or for users to protect their sessions after they are finished using the application. Upon reception of this message the authentication daemon notifies the SSA worker threads, which in turn destroy all session and ticket data associated with the TLS connection, and send a TLS `close_notify` to the server.

### 4.8.3 Authentication Device

We implemented an authentication device as an Android mobile application for smartphones. The application, named "Securely", stores and manages user credentials, performs registration and login actions when authorized by a user, and stores identifying characteristics of authentication daemons to which it connects. Our implementation makes use of the Android keystore system, which protects keys from application processes and extraction from the device.

Figures 4.9 and 4.10 shows a progression of screenshots from the Securely app as displayed on a smartphone. During first-time setup, the application provides a brief introduction and prompts the user to provide basic personal information, such as a username, real name, and email address (Figure 4.9a, 4.9b). The values input by the user are used as the defaults for certificate generation. Users can override or omit these when responding to future registration requests. After this setup, and whenever the Securely application is launched thereafter, the user is asked to select the SSA-enabled host to which the app should connect (Figure 4.9c). This list is populated using the Android Bluetooth API for Bluetooth services, and Network Service Discovery (NSD) for multicast DNS in Wi-Fi environments. After a host is selected, the application connects to it via TLS, using connection information broadcast during discovery. If the application does not recognize the public key presented in the connection, it requests that the host display its public key in QR code format on its physical screen for verification. The application then requests the user to scan the QR code, which is done automatically as the user focuses the phone's camera on the host screen (Figure 4.9d, 4.10a). If the public key provided during connection matches the one scanned from the user, the key is remembered for future connections and the user is prompted to proceed to use the target host machine (Figure 4.10b). If the public key does not match, the user is shown a warning on both the application and host screen that the connection is insecure and the application will abort the pairing. Figure 4.11 shows the security messages temporarily displayed by the host for both success and key mismatch cases.

Once securely connected to the host computer, the application listens for authentication requests from the host machine. This listening will continue until the connection is severed between the two devices, which occurs when the phone leaves the area (or network), or if the user chooses to manually disconnect. Throughout the duration of this connection, requests for TLS client authentication will result in the user being prompted for consent, as shown in Figure 4.10c and Figure 4.10d. If the application is not in the foreground (e.g., due to a

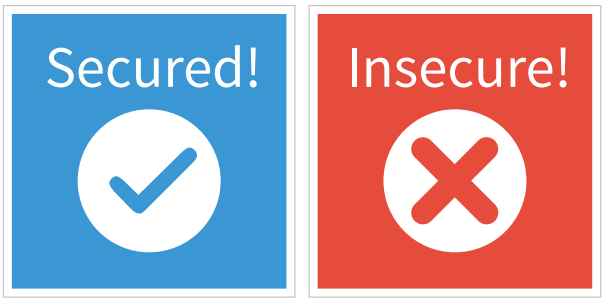



Figure 4.11: Temporary splash images shown on the host machine after a user connects a remote authentication device. Left: success case. Right: key mismatch case

chat application being active), the user will receive a similar, interactive notification through which a course of action can be selected.

Since TLS sessions may be shorter-lived than the amount of time a user is actually using a service, users can configure how long the application should automatically respond to login requests. This is similar in nature to an HTTP login cookie, except that its expiration is set unilaterally by the user, allowing greater user control over login duration. Users can also perform an asynchronous logout, using a logout request message functionality, which causes the application to stop automatically responding to login requests and causes the host machine to delete the current TLS session for the relevant application.

It is important to note that registration requests are not explicitly part of the protocol between the authentication daemon and connected devices. Instead, authentication devices interpret authentication requests for certificates it does not have as registration requests. In these cases, if authorized by the user, it first generates or obtains a certificate for that host, and then responds to the underlying authentication request using the new certificate. While a list of certificates could have been provided to the authentication daemon to help it distinguish between requests for registration and authentication, this would require users to disclose all of their certificates to the host, which has undesirable privacy implications.

## 4.9 Discussion

Through separation of the security and application layers, the platform described addresses many of the standing problems in TLS client authentication. The security layer is handled by the operating system, which allows applications to easily access TLS authentication functionality, while simultaneously increasing user privacy by preventing applications from gaining too much visibility into the authentication process. The benefits of this separation can be cataloged from the perspectives of servers and administrators, as well as clients and users.

### 4.9.1 Benefits for Servers and Administrators

Many of the issues described by Parsovs concern the difficulties of the server components of TLS client authentication deployments. We find that these are due to the complexities of mixing application logic with TLS functionality. The simplified API hardens the use of TLS client authentication within server applications. We outline some of examples of this with respect to the issues mentioned by Parsovs.

1. **Server development and configuration:** The first issue brought up by Parsovs is that of the difficulty in configuring Apache's `mod_ssl` for enabling TLS client authentication. In this case the configuration directives are incorrectly documented and have subtle semantics that may be confusing to administrators. While this work does not supply a new webserver configuration platform, we do extend the SSA to easily support client authentication. Using the new socket options, we have integrated TLS client authentication into an SSA-enabled webserver, allowing it to easily request client authentication under a variety of configurations. These features allow trust-chain authentication by specifying a set of trusted authorities, and CA-less authentication by accepting self-signed certificates.

Parsovs also noted two issues with certificate validation on the serverside. The first was that full-chain validation is not performed, forcing applications to perform some validation on their own. Complicating this further is the fact that the full client chain is not exported to web applications. Second, it was noted that advanced certificate validation checks were not performed, such as certificate revocation checking. Due to our use of the SSA, applications benefit from both full-chain validation as well as the ability for administrators to fine-tune how each application performs certificate validation.

The SSA exports the full certificate chain offered by remote peers with a single function call, to applications who wish to perform more custom types of validation. We expose this data to PHP and other web application languages by forwarding SSA socket information to CGI environment variables. This information can also be used to properly audit authentication failures, as errors can be properly exposed to the application layer.

2. **Information leakage:** Parsovs noted that under OpenSSL and `mod_ssl`, client authentication validation is performed before the client has proven ownership of the certificate it transmitted. This leads to a security vulnerability wherein attackers can gain some information regarding the trust settings of the server, as they can submit arbitrary certificates to the server and observe whether it aborts the TLS handshake. Under our platform, the request for authentication by a client is an atomic operation, and handled by the operating system. That is, I/O is suspended from the perspective of the application from the time that a `CertificateRequest` is sent, until reception and validation of both `Certificate` and `CertificateVerify` messages. Applications gain this feature implicitly, with no effort required on behalf of the developer, which prevents application logic from interfering with security logic.

3. **Information procurement:** Perhaps most troublesome was Parsov's finding that server applications using TLS client authentication had difficulty gaining information about the freshness of a client's proof of identity. In the case of TLS 1.2 and below, the

freshness of a client's proof of identity can be measured from the time the `ServerHello` is sent by the server, as that is the first unpredictable handshake transcript message that the client must sign when crafting its `CertificateVerify` message. However, existing TLS server libraries do not export any information that would allow this measurement. Furthermore, Apache was found to reset TLS timeouts every time it received any new data from the client. Together, these issues can provide attackers with an arbitrary amount of time to craft their own false `CertificateVerify` message, which may take as little as a few hours [9]. A related issue mentioned by Parsovs is the inability of server applications to communicate with `mod_ssl` to request that it invalidate the authentication, making access denial problematic.

We avoid these issues by providing timeout behavior to client authentication at the socket level, within the operating system. When an application has requested client authentication, I/O operations on the client socket are suspended, to prevent accidental disclosure of confidential information or mislabeling data as sent by an authenticated party when it is not. If a proper `CertificateVerify` message has not been received after the default socket timeout period, a TLS timeout will occur, regardless of any other data transmitted by the client. In addition, applications can request that the platform deauthenticate a client by setting the TLS session timeout to zero using a call to `setsockopt`. Using these features, applications can obtain assurances about the authenticated state of a client and simplify their authentication logic.

### 4.9.2 Benefits for Clients and Users

Developers of client applications and users are also benefited by the separation of the security and application layers. Client applications obtain automatic support for client authentication and users are afforded greater privacy, security, and flexibility when using strong authentication methods.

1. **Built-in client authentication support:** The primary benefit for client application developers is that by using the platform, they do not need to concern themselves with writing code for indirect client authentication, as it is handled by the operating system. As a result, login systems using strong authentication can be supported effortlessly, even if a developer has no knowledge of public key cryptography. The services provided by the authentication daemon are invoked automatically, as a result of the remote server requesting authentication. This effectively reduces the amount of client code required to implement a strong authentication system to zero. If developers would like to utilize direct authentication, specifying specific keys and certificates to use, they can do so using the relevant SSA endpoints, calling `setsockopt` with options to set the certificates and keys to use.

2. **No private key transport:** Although not directly a result of the separation of layers, the decision to provide an authentication daemon that does not request access to user's public keys allows users to be more secure, and use devices that prohibit the disclosure of private keys for authentication. This protects keys from attackers who compromise an application and attempt to access key material, or malicious applications attempting to steal keys. This is in line with the principle of least privilege, as applications do not need to access the key directly to format and send a `CertificateVerify` message.

3. **Greater privacy:** While the TLS protocol itself has a need to transmit a user's certificate, applications do not need to access it directly. Also in line with the principle of least privilege, the platform places certificates in the hands of the operating system, and transparently forwards it to the intended remote host. In doing so, the underlying TLS implementation also ensures that the certificate is not transmitted in the clear, further protecting user privacy. That is, authentication will only be performed after a secure renegotiation under TLS 1.2 and below. These two enhancements allow users to authenticate to remote hosts without trusting the client application with their identity. This feature may be particularly useful to those residing in areas with weaker freedom

of speech, or in cases where users want to avoid applications that use their data for tracking and advertisement purposes. While a compromised operating system could still expose the certificate to attackers, we assume this to be a more difficult task, especially in the presence of modern kernel hardening mechanisms such as Windows Device Guard and driver signing.

4. **Unification:** Using the authentication daemon, the operating system supports a unified experience for both applications and users when servicing client authentication requests. The daemon is agnostic to the physical transport mechanism used to communicate with authentication devices, and supports software interfacing as well. Due to this centralization, the user interface provided by the authentication daemon is common to all applications using the platform, providing consistency for users with their authentication device of choice.

5. **Enhanced functionality:** The centralization of authentication daemon services also allows authentication devices to be provided with standard authentication functions beyond attestation. Devices can set authentication session duration, which allows their devices to automatically respond to repeated authentication requests from the same remote host and application. They can also opt to asynchronously terminate their TLS sessions, through the use of a protocol message that forces the SSA to terminate the underlying connection and remove any saved session state. These options are automatically built into all client applications, eliminating the need for developers to support them explicitly. Another benefit is that the authentication daemon can be configured to automatically terminate sessions when an authentication device disconnects, preventing users from accidentally remaining logged on to their accounts in public settings. The standardization of these features also allows users to find them more easily, as they only have to become familiar with the UI of their device to use them.

## 4.10 Beyond Authentication

Post-handshake authentication (PHA) is a new feature of TLS with version 1.3. PHA allows servers to request that the client authenticate at any time, during an established TLS connection, without the need for additional handshakes. This provides a natural way for authentication to be refreshed. In this work, we explore this feature's applicability to two other areas: sensitive information transmission and authorizations.

In the `CertificateRequest` message, the server can indicate to the client which certificate to use in a reply by optionally specifying a list of Distinguished Names (DNs) representing the names of preferred issuing authorities. In addition, or instead, the server can use the `oid_filters` extension in the `CertificateRequest` message, which specify a series of key-value pairs that the client certificate should match. Under PHA, the server can send multiple `CertificateRequest`s in a single session, and modify the filters used, as desired. In response to each, the client must send an appropriate `Certificate` and `CertificateVerify` message. This allows the client to easily authenticate to the server in multiple distinct ways throughout a single TLS connection.

Login credentials like passwords are not the only sensitive data provided by users when using network applications. Often, other information such as mailing addresses, credit card numbers, and phone numbers may be requested as well, such as during an ecommerce transaction. Currently, these data are visible by the client application, and also subject to keylogging and other eavesdropping processes that may be present. PHA, in conjunction with the operating-system managed TLS authentication in this work, offers a compelling avenue for the private transmission of such data. Rather than requesting sensitive information at the application layer, server applications can send a `CertificateRequest` that tells the client to authenticate using a certificate containing specific fields of information. Under the platform described, this information would not be visible to the client applications, being forwarded directly by the operating system to the server. Different information can be specified with different `oid_filter`s, and the certificates sent by the client that contain this information can

be generated once and self-signed. This use of certificates may be useful for users who are wary of entering information on public computers, such as credit card numbers. This mechanism also allows users to obtain functionality similar to saved HTML form data supported by modern browsers, but without tying this information to a local installation or cloud account. Storing the certificates containing this extra information on an authentication device also allow users to manage all of their information from a single interface, like a digital wallet.

Certificates containing sensitive information need not be self-signed, and there are some benefits to having such information be signed by the parties who issued or validated it. For example, ecommerce transactions could be more simply validated if users were issued certificates from credit card companies that contained their billing information and card numbers. This has the added benefit of users not having to provide this information manually. The use of such certificates through an authentication device can be seen as a form of authorization, rather than authentication. Under this scheme, servers could validate client certificates against one or more trusted signing certificates owned by credit card companies, and present the associated names in the DNs for a `CertificateRequest`. We briefly explored user interactions with a simulated certificate-based credit card in a user study described in Section 4.11. Another use of this feature is to subvert session hijacking, as it requires that users confirm transactions through their authentication device. A similar technique was used by MP-Auth password-hardening system which leveraged mobile phones, for "transaction confirmation" [54] with ATMs and applications.

## 4.11 User Study

We conducted an IRB-approved user study to obtain insight into the usability of our smartphone-based authentication device, the usability of connecting to the authentication daemon, and user opinions about using the system for authentication tasks. In this study, 20 individuals participated in a short role-play as they used the Securely application on Android smartphones to connect to a desktop computer, and perform various authentication tasks. In

this study, participants were asked to use the desktop computer for web browsing, connecting to a mock ecommerce website and performing registration, login, and checkout. Participants were compensated $10, and each study was conducted within 30 minutes.

When participants arrived for the study they were told to assume the role of an online shopper who wants to purchase a pair of shoes. They were also instructed to use the phone and desktop computer provided to them as if these were their own. We then provided them with a list of tasks to perform, which included:

- launching the mobile application and following its instructions (during this step, the mobile application walked participants through the process of setting default information and connecting to the authentication daemon for their computer, shown in Figures 4.9a – 4.10b).
- registering and logging into an account at the ecommerce site using a desktop browser
- browsing the site for any pair(s) of shoes desired and adding them to the shopping cart
- completing the purchase via checkout on the website

As users browsed the site, the mobile application reacted to various messages from the authentication daemon running on the computer. When users opted to register for an account with the website, the application prompted users for registration as shown in Figure 4.10c. Any subsequent login attempts caused the application to prompt users as shown in Figure 4.10d.

In addition to using the mobile application for basic authentication tasks, we leveraged TLS 1.3 PHA, as discussed in Section 4.10, to prompt participants for checkout using the mobile app. In this case, the server requested PHA from the client, asking for a credit-card certificate instead of the user's login certificate. When this occurs, participants see a prompt similar to Figure 4.10d, but the prompt language is modified to appear as shown in Figure 4.12. The app also notifies the user of the information that will be shared to the remote host upon acceptance of the purchase, which includes credit card and billing address information. For the purposes of this study, a default certificate with dummy information was preloaded

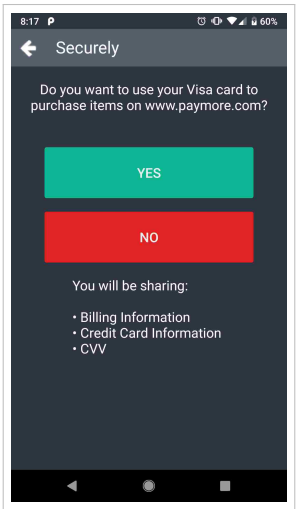



Figure 4.12: Credit Card Authorization Prompt

onto the mobile phone for use by participants beforehand. The checkout process on the website allowed users to choose between classic checkout methods and using an attached authentication device, but no participant opted for the former.

After finishing their tasks, participants were asked to complete a survey. The survey asked participants to rank their experience using the application with the System Usability Score [13] (SUS), provide information regarding their experience, compare the system to web password authentication, and report some demographic data. Appendix A contains the task sheet and survey provided to participants.

The desktop computer was configured with the SSA and default settings, and a Chrome web browser that used the SSA, with our extensions, for its TLS connections. The browser connected to a remote server that hosted the mock ecommerce website and ran a custom webserver that uses the SSA augmented with our TLS client authentication API. This webserver was chosen because at the time of experimentation, common webservers did not support TLS 1.3, and we also used the study as an opportunity to test our server-side authentication API in a practical setting.

The mobile application was reset for each study session, allowing each participant to perform the first-time setup process.

Participants were recruited on a university campus through the placement of fliers in high-traffic areas, but not in areas common to those in computer-related disciplines. The resulting 20 participants were fairly evenly split across gender (Female: 11 (55%), Male: 9 (45%)). Due to our recruitment on a university campus, the majority of participants were students (18; 90%) and fairly young, most being 18-24 years old (18; 90%) and the others being 25-34 years old (2; 10%). Participant area of study/occupation was very diverse, with 19 distinct fields represented. The level of computer expertise reported by participants was largely "intermediate" (16; 80%), with two falling between intermediate and "beginner" (2; 10%) and two between intermediate and "expert" (2; 10%).

### 4.11.1 Results

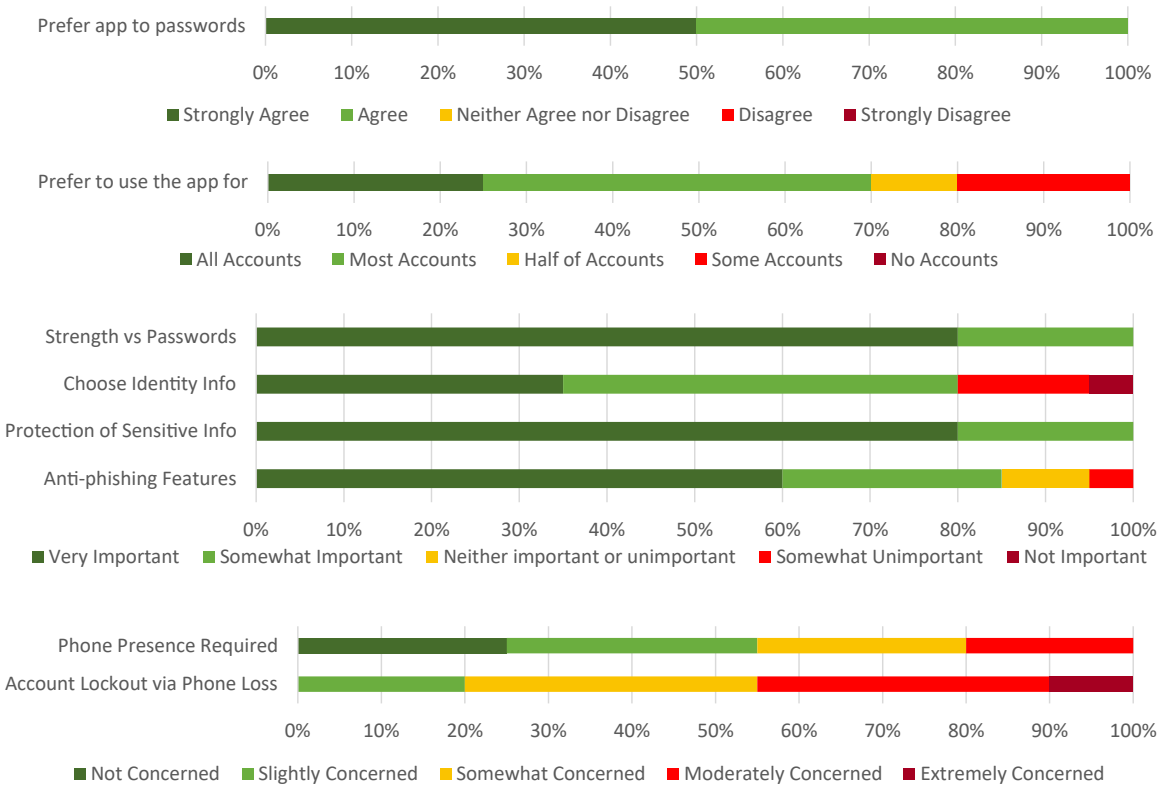



Figure 4.13: Post-study Likert responses from user survey

All of the participants were able to successfully complete the assigned tasks. This included connecting to the authentication daemon using the smartphone application, and scanning the host QR code to authenticate that connection. However, we note that the smartphone was already connected to the local network via WiFi. This was intended to mimic a home network scenario, in which users likely already have their smart device connected to their local WiFi connection.

Across the 20 participants, the overall system achieved an average System Usability Scale (SUS) score of 87.4, well-within the "acceptable" range and between the "good" and "excellent" adjectives.

We also asked participants a series of additional questions after the standard SUS questions, the results of which are shown in Figure 4.13. When asked if they would prefer to use the mobile application instead of passwords, all participants selected "agree" (10; 50%), or "strongly agree" (10; 50%). Participants were then asked to quantify how many of their accounts for which they would prefer using the application instead of passwords. Responses to this question varied, with some participants selecting "some of my accounts" (4; 20%) and "half of my accounts" (2; 10%), many selecting "most of my accounts" (9; 45%), and finally some selecting "all of my accounts" (5; 25%).

After providing this information, participants were informed of four features and two limitations in the use of the application that could not be adequately introduced by the role-play. They were then asked to report how important these features and limitations were to them, on a 5-point Likert scale. First, participants were told about the strengths of digital signatures versus passwords, that passwords were not being used by the system behind the scenes. Most participants said this feature was "very important" to them (16; 80%) and the remainder said it was "somewhat important" (4; 20%). Second, participants were told about the ability to choose the information presented to hosts during registration and login, which mentioned the ability to register with false information. Participants were less interested in this feature, with some indicating that it was "not important" (1; 5%) or

"somewhat unimportant" (3; 15%), but most indicating that it was "somewhat important" (9; 45%) or "very important" (7; 35%). Participants then were told about the system's ability to protect information other than login credentials from an application's view, as demonstrated by the secure checkout using PHA and a credit card certificate. The responses to this feature were the same as the first, with most participants indicating that it was "very important" to them (16; 80%) and the remainder indicating "somewhat important" (4; 20%). Finally, participants were told about the system's resistance to phishing (due to the application's protection of private keys and verification of account presence before login). Participants reported less strongly to this feature, with some reporting it "somewhat unimportant" (1; 5%) and "neither important or unimportant" (2; 10%), and the rest reporting it "somewhat important" (5; 25%) or "very important" (12; 60%).

Next, participants were told about two limitations of the application, and asked to report how concerned they were by them. First among these was the requirement that participants have their smartphone with them to access the accounts it manages. Participants were very split on this issue. While none of them reported to be "extremely concerned" by this, all other options were represented: "moderately concerned" (4; 20%), "somewhat concerned" (5; 25%), "slightly concerned" (6; 30%), and "not at all concerned" (5; 25%). Also mentioned was the limitation that lost phones would effectively lock users out of their accounts, as the credentials within them cannot be transferred. However, it was also mentioned that we intend to eventually support a credential reset option. Participants were slightly more concerned about this issue: "extremely concerned" (2; 10%), "moderately concerned" (7; 35%), "somewhat concerned" (7;35%), "slightly concerned" (4; 20%).

After learning of these features and limitations, participants were asked if their opinion of using the application versus passwords had changed. A little over half indicated that it had (11; 55%). If participants selected this, they were again asked to rate their preference for the application versus passwords and to re-quantify how many of their accounts they would like to use with the application. Of those that opted to change their preference, three participants

downgraded their preference for the application, one increased it, and the remainder did not change from their original response. Five upgraded the number of accounts they wanted to use with the app, four downgraded, and two did not change from their original. Overall, after these preference changes, two participants no longer favored the application versus passwords, but the remainder did (40% for "agree" and 40% for "strongly agree"). In addition, the number of accounts that participants wanted to use with the applications stayed roughly the same (20% for under half of accounts, 65% more than half of accounts).

A complete listing of all the anonymized results can be found at `owntrust.org`.

### 4.11.2 Analysis

The SUS score of 87.4 is encouraging. We note that it is difficult to distinguish between participant opinions regarding the smartphone application and the functionality of the authentication daemon, given that one cannot provide its function to users without the other. However, the high SUS score demonstrates that use of these technologies together provide a strong avenue for usable strong authentication. This is particularly fortunate, given the fact that ownership of mobile devices is widespread and users typically have them on their person, mitigating the issue of key portability. In addition, a usable authentication daemon provides users a common interface for performing authentication for any network application on their computer.

The high preference for the application versus passwords and the heavy skew toward using it for most accounts is also encouraging for researchers engaged in replacing password authentication. When asked what kinds of accounts they would use (and not use) the application with, some participants mentioned all their accounts (5; 25%), others mentioned shopping and online media accounts (7; 35%), and others mentioned accounts for which they often forget passwords (2; 10%). Eleven participants mentioned that they would be hesitant to use the application for banking or "sensitive" accounts, with two explicitly mentioning they want to know more about how their information was secured before its use in that environment.

However, unprompted, eight of these rescinded this hesitancy after completing the portion of the survey that further informed them of the application's features and limitations. The others did not mention it further. This sheds light on the state of user education regarding the relative strengths of passwords versus stronger authentication methods. At least in this study, some users had a natural aversion to the departure from well-known password authentication, but were more trusting when informed of the benefits of password replacement technology.

Three users lowered their preference toward their use of the application when faced with the additional information presented in the survey. In one of these cases, we suspect that it was an error, as the reason provided for the downgrade was that the additional information made the participant "more comfortable" using it with sensitive accounts. However, in the other two cases, participants explicitly mentioned their concerns over lost, broken, and stolen phones rendering their accounts inaccessible, also citing that a credential reset system would alleviate these concerns.

Another interesting datapoint of note is that all participants valued the ability to keep other sensitive data, in addition to login credentials, out of the purview of applications like browsers. This understanding may assist developers of authentication devices to also consider the storage and secure transfer of values other than credentials.

## 4.12 Future Work

Numerous avenues for future work have emerged as a result of this exploration. Some of these enhance the work described directly, and others propose advancements that benefit authentication technologies in general.

### 4.12.1 Renewal and Recovery

As highlighted in our design and user study results, account recovery after lost or stolen private keys is an important issue to resolve. While renewal and recovery can and are handled by various certificate authorities and organizations in their own ways, a more standard

and automated mechanism would facilitate user adoption. Recently, automation of server certificate issuance via challenge-response has been popularized by Let's Encrypt [45]. Using this service, server operators can obtain a trusted certificate for their websites for free, with server identity attestation taking place through "Automatic Certificate Management Environment" (ACME), a simple challenge-response protocol. This obviates the need for server owners to pay for certificates and follow tedious, manual steps to authenticate themselves to the certificate authority.

While the automatic nature of this issuance is subject to its own security concerns [11], the Let's Encrypt service provides certificates for more than 115 million websites as of August 2018 and its root certificate is trusted by all major trust stores [1]. We believe that automated, free certificate issuance can be extended to users. In this case, certificates can be issued following a series of responses to challenges of identity attributes, such as phone numbers or email addresses. For example, authentication devices could create certificate signing requests on a routine basis, and send them to one or more "Let's Authenticate" services. These services could provide an automatic challenge-response protocol similar to ACME, that required users to prove identity, and offer short-term signed certificates after acceptable proof is received. For example, to prove ownership of an email account, an authentication device could use user credentials to automatically receive and respond to an email challenge to obtain a signed certificate from the service that attests ownership over that email address. Similar methods could be instituted to prove ownership of social media accounts. Operators of small services who lack the resources needed to implement their own PKI architecture could then selectively trust these issuers in their TLS server applications. This would obviate the need for them to handle their own certificate issuance, renewal, and revocation - akin to the functionality of Single Sign-On for web password authentication.

### 4.12.2 More User Studies

Our work provided only a preliminary user study. This study served as an initial probe into the use of a central authentication daemon by users with external authentication devices, demonstrating its feasibility and attractiveness to users. While this is a good first step, many different studies can and should be executed to further develop usable tools for TLS client authentication. Our study mimicked a typical home setup wherein a user connected to an authentication daemon on his own computer. One avenue for future study is to evaluate usability in the context of public WiFi or enterprise work environments. In addition, evaluation of user reactions to man-in-the-middle attacks between the authentication device and daemon would also be helpful to ensure proper user response and understanding. Longer-term studies would also be greatly beneficial, to evaluate authentication strategies for repeated use over the period of days or weeks, rather than a single session in a laboratory. Finally, studies could also be performed to determine the boundaries of user comfort with respect to the generation and use of certificates for non-authentication scenarios, such as ecommerce checkout.

### 4.12.3 New Features

This work strictly explored TLS client authentication using certificates. However, it could easily be extended to support other forms of TLS authentication, such as TLS Secure Remote Password (TLS-SRP), and pre-shared keys (PSK). Rather than generating a keypair and certificate upon registration, the authentication device could prompt the user for a password to use with the remote host. During authentication, the user could either enter in the password on the device or use a cached copy, but the password itself would never leave the device. Since passwords are known to users, this feature would have the added benefit of more portability, allowing users to use multiple authentication devices without requiring any keystore synchronization.

Some stronger authentication methods, such as TLS-SRP, use passwords to function, yet have the ability to thwart credential-stealing phishing schemes if implemented properly. A major problem in deploying these methods is that they tend to rely on UI elements from the application, such as a browser, both within and outside of the application chrome. In these cases, attackers can still intercept user credentials by mimicking the look and feel of input fields, a problem previously noted by McCune et al. [59]. This can be thwarted if the user only enters sensitive information in protected UI areas. In this work, the protected UI came in the form of a second screen and device, the smartphone connected to the platform. Since a web browser or other application does not have permissions to edit the UI of such a device, attackers cannot attempt their UI-emulation in this case. However, in cases where an external authentication device is not desired, the user does not have a protected UI area. Fortunately, the SSA provides an avenue to address this weakness, as it can leverage the operating system. Using this privilege, a secure password entry UI could be created and protected that would prohibit applications from obtaining information that was entered when the user is focused on a specified portion of the screen. To ensure robustness, this undertaking would require a fair amount of interfacing with display drivers and operating system windowing services.

## 4.13 Limitations

Our approach to TLS client authentication has some limitations. First, the issues of certificate renewal and revocation are not directly handled by the solution proposed. While enterprises already using TLS certificate authentication likely have existing methods to handle these cases, the absence of such functionality from the solution precludes the use of TLS client authentication as a general password replacement for the masses. Solutions that may solve this issue are the proposed Lets Authenticate system, systems supporting self-sovereign identity, or other systems that streamline the process of identity management for ordinary users.

Another limitation of the system is the need for users to have their authentication devices within reach if they wish to authenticate. This requirement can be particularly frustrating for users if, for example, they leave their phone at home and need to authenticate at work. While hardware-based two-factor authentication methods have the same drawback, this nonetheless may discourage users from adopting such a technology for all the accounts with which they would otherwise wish to use it. That said, we expect that the use and toting of smart devices will becoming increasingly ubiquitous in the future, which may mitigate this concern to a large degree. For those unable or unwilling to always have another device on their persons, cloud-based software authentication devices may be a compelling option. However, such solutions would require some other token, such as a password, broadening the attack surface of the system.

TLS terminator compatibility may need to be addressed in light of our system. In cases where a datacenter uses TLS terminators near its gateways and communicates with internals servers over a separate connection, client authentication information may be discarded before it reaches the backend server. In these cases, terminators should communicate relevant client identifiers to backend servers within the application layer protocol (e.g., insertion of HTTP headers) or through some other channel.

Finally, the proposed solution requires client applications to use the SSA. While we advocate the inclusion of the SSA or similar systems within operating systems, these have not yet gained widespread use. To serve as a stopgap adoption measure, and to allow early adopters to use our system, our solution includes a local transparent proxy that forces network applications to use the authentication daemon when remote servers request TLS client authentication. This proxy works with both command line utilities and GUI applications, such as Firefox and Google Chrome, and was used in our user study.

### 4.14 Use and Integration

Long-term we would like to see PKI services like Lets Authenticate or self-sovereign identity networks created and deployed to provide easy access to trusted credentials. This would greatly enhance the deployment surface for the technologies we present. However, there are already many compelling deployment opportunities. While TLS client authentication is not used broadly by ordinary users, it is employed by some enterprises and is supported by a wealth of server software. Recently, Cloudflare has added support for TLS client authentication for all of their service plans [37]. While Kerberos is widely used in intranet environments, it is rarely used by services on the public Internet. As a result, VPN logins, external access to enterprise mail, and other services are usually accessed via TLS and passwords. As cloud enterprise services continue to gain popularity, cloud enterprise customers may find our approach advantageous for their strong authentication needs. Leveraging our platform, administrators can force local client applications to use the authentication daemon for a unified experience that supports all TLS applications. Likewise, in-house developers of such enterprises can rely on the SSA and authentication daemon for their security needs and focus exclusively on application functionality. For enterprises using Kerberos, our solution may also find use as a strengthening technology for authentication of clients with the KDC via PKINIT, which enables the KDC to use X.509 certificates for mutual authentication. This would provide employees with the same login experience both locally and remotely.

Despite our focus on TLS client authentication, It is our hope that OS vendors integrate a variety of authentication services into their platforms, to provide applications with transparent, flexible authentication support. This move also empowers administrators to better configure security to their needs. Support could be expanded to incorporate authentication at other layers of the protocol stack as well, such as WebAuthn, or even local authentications, such as operating system login. The standardized CTAP protocol would benefit from integration with the operating system, allowing applications to rely on it as a service rather than implementing it on their own. Furthermore, users would be benefited by

authentication devices that were protocol-agnostic, unifying the UIs of authentication for all protocols and methods.

## 4.15 Related Work

Parno et al. [70] and PhoneAuth [17] both present efforts to strengthen authentication by leveraging a mobile device connected to a host computer. Work by Parno et al. is focused on preventing phishing attacks. Users initiate TLS connections from their phone, and a provided web browser extension performs TLS client authentication with a self-signed certificate from the phone, as a second factor to a primary password. PhoneAuth also serves as a second-factor authentication, transparently supporting channel binding via BlueTooth communication with a user's phone during website login. In contrast with these works we seek to employ TLS client authentication as a primary authentication mechanism, and do so in a manner that increases usability for developers as well as users. We also leverage the operating system to provide application-agnostic support and increased privacy.

Pluggable Authentication Modules (PAM) is a framework that decouples authentication mechanisms from application authentication requests. Using a single API, developers can obtain authentication functionality provided by one or more authentication strategies installed and configured by an administrator. This work is similar in spirit to this effort, as the client additions to the SSA essentially decouple the mechanism from the requesting application. However, our focus is on TLS specifically, and we support authentication without an explicit API, instead relying on an operating system based TLS implementation.

There is a large body of work concentrated on replacing passwords as the primary means of user authentication with stronger alternatives. Bonneau et al. provide an overview of this body of work, noting the difficulties each approach encounters when pit against passwords [12]. This body includes various mechanisms, from those that still leverage passwords in some way, to alternative methods that replace passwords entirely with keys. Two-factor authentication mechanisms (2FA) often use passwords as their first factor, and

federated protocols like OpenID [76] limit the number of passwords a user needs to remember. Key-based mechanisms such as certificate authentication and PGP do not use passwords, although one exception to this is the family of password-authenticated key agreement (PAKE) protocols. Many implementations of the mechanisms mentioned exist, and some vendors support a variety of these with a single hardware token, such as Yubico [94].

The Fast Identity Online (FIDO) Alliance is a consortium that produces open, general specifications for strong authentication mechanisms to increase interoperability among key-based authentication mechanisms [3]. The FIDO Alliance has numerous open standards, including the Universal Authentication Framework (UAF) and Client-to-Authenticator Protocol (CTAP). These standards specify the terminology, message structure, and other technical details that implementers of strong authentication methods can employ to be interoperable with other technologies. Specific to browsers, WebAuthn, is an API defined by the W3C to enable strong authentication for the web, allowing websites to utilize JavaScript endpoints to invoke browser authentication features [87]. Microsoft, Google, and Mozilla have already begun integration of WebAuthn with their respective browsers.

Another improvement to password authentication is PAKE protocols. This family of protocols provides mutual authentication based on a shared password between two or more entities, with a primary benefit that the password itself is never actually sent to the server. Instead, parties communicate a proof of knowledge of the password, which leaks minimal information to an eavesdropper. Of these protocols, a subcategory of Augmented PAKE protocols operates in a client-server environment that additionally allows for no password-equivalent data to be stored on the server. The Secure Remote Password Protocol for TLS (TLS-SRP) [86], is a TLS extension using an Augmented PAKE protocol and is supported by OpenSSL. OPAQUE builds upon Augmented PAKE, offering a stronger formalism and mitigating its vulnerability to pre-computation attacks wherein attackers can instantly recover user passwords after populating a password dictionary and compromising the server [46].

## 4.16 Conclusion

This work explored the creation of an operating system platform for TLS client authentication. This platform facilitated the deployment and configuration of TLS client authentication for server and client software, allowing developers to implement it with as little as one line of code on the serverside, and zero on the clientside. Furthermore, this platform allows private keys to remain within the confines of remote devices, and even isolates certificates from applications, increasing privacy. We also discussed the potential of the post-handshake authentication to support features beyond authentication, finding it a compelling avenue for the transmission of sensitive information as well as user authorizations and deploying it in a user study. To evaluate our efforts, we deployed TLS client authentication on an SSA-enabled webserver, constructed an authentication device application for Android smartphones, and conducted a user study to assess usability. The user study revealed that users are capable of using the authentication daemon for indirect TLS client authentication and that they view the authentication application as a favorable alternative to passwords. In addition, we find that users are welcoming of the features that protect more than login credentials. However, we note that much work has to be done to test and develop authentication devices to meet user needs. This is especially true for account recovery mechanisms, and we suggest systems for automating this process.

# Chapter 5

# Future Work

We have seen the works described in this dissertation leverage the power of the operating system to solve outstanding problems in network security for TLS. These works have delineated the roles of developers, administrators, and users, unburdening each from misplaced responsibility and empowering them to easily strengthen security. As a result of this effort, we have uncovered numerous avenues for future work. In this chapter we discuss these, noting areas where the motivating principles of this work can be furthered to:

- enhance security beyond the scope of TLS,
- functionally extend the solutions discussed in this work, and
- supply recommendations to practice and protocols.

At the end of this chapter, we also provide some high-level discussion regarding the benefits and drawbacks of centralization of security, and a discussion regarding the possibility of conflicts between developer and administrator policy. For each of these, we also note avenues for future exploration.

## 5.1 Application Beyond TLS

In this section we discuss the application of the motivating principles of this work to domains beyond TLS. This includes securing hostname lookup, via the Domain Name System (DNS), through operating system services, and supporting other internet protocols.

### 5.1.1 Securing DNS

TLS is only one area of security that can benefit from operating system control. The Domain Name System (DNS), and its associated protocol, is another potential avenue for exploration. DNS security is tightly coupled with TLS security in at least three ways. First, X.509 server certificates are validated in part by verifying that the hostname of the intended server matches that of the provided leaf certificate. Typically, this hostname is first queried against the DNS to obtain the IP address of the server. Second, TLS privacy is linked to DNS security. While TLS 1.3 has an encrypted handshake that protects the server hostname requested by the client, the client must still resolve this hostname to an IP address beforehand with a DNS lookup. Finally, some certificate validation strategies, such as DANE [39] and ConfiDNS [72] leverage DNS records to operate. Securing DNS is therefore advisable for those interested in holistic network security.

Centralized services that facilitate the secure use and deployment of DNS-securing technologies could be designed to help both clients and servers. Client applications, currently relying on standard DNS lookup functions such as `gethostbyname` and `getaddrinfo`, would be well-served to have an operating system service properly initiating these queries using DNS over TLS, and performing validation on any security-related records returned. Such a service could then also integrate into TrustBase and the SSA, providing them with additional validation information. This service could additionally support anti-phishing efforts, using the hostname as input to check for imposter domains. Server applications could benefit from a similar service, which could automatically create DANE records using the server's private keys. This would be particularly useful in conjunction with the SSA and services such as Let's Encrypt.

### 5.1.2 Supporting Other Protocols

TLS is not the only network security protocol, and it may not always be the most popular. However, we believe the interfaces exposed to administrators and developers by the SSA are

fairly generic, and that the POSIX socket API is capable of supporting network security protocols in a generic way. To test this theory, support for modern protocols, such as QUIC, could be added to the SSA. Since QUIC merges and modifies some traditional layers of the networking stack such as UDP, TLS, and HTTP, its integration would further test the flexibility of the POSIX socket API.

## 5.2 Extending Presented Solutions

In this section we discuss extending the individual works described in the preceding chapters. We discuss feature additions that allow consent-driven TLS traffic interception, anonymity service support, leveraging new hardware capabilities, integration with existing standard protocols. Also noted are avenues for extending the work for our secure login application and related infrastructure.

### 5.2.1 Consent-driven key sharing

Numerous enhancements to the work described in this dissertation are possible. One area for improvement is leveraging the SSA to tackle the longstanding issue of TLS inspection. TLS inspection is the practice of organizations performing TLS MITM attacks on their own networks for the purpose of protecting the organization from information leakage, malware, and other threats. Both hardware and software products are sold to provide this service, and its use in practice is a matter of debate among security experts. On the one hand, these products protect the organization's machines and network. On the other, they do so in a way that violates the end-to-end security guarantees of TLS and violates the privacy of users. Previous work has found that while the majority of users are accepting of the practice in general, overwhelmingly they would also like to be notified and asked for consent when it occurs [79].

The development of the TLS 1.3 specification renewed tensions concerning this practice, as elements of the new protocol threatened to undermine some of the mechanisms that made

traffic inspection possible and practical. This prompted the creation of proposals that extend TLS 1.3 to allow the client and server to grant access to a monitoring third-party, such as the TLS Visibility Extension [43].

The SSA could be extended to satisfy both user privacy and the security needs of enterprise administrators. Since the SSA has access to the master secret for each connection, it is capable of permitting third parties to gain visibility into any data stream. This can be done completely outside of the TLS channel itself, without requiring modification to the protocol, potentially simplifying the design of protective network products. The SSA presence on the local machine also permits it to notify users and obtain their consent before this takes place. An important piece of this effort would be the task of ensuring that UI notifications are designed in ways that make users fully aware of the consequences of inspection and able to easily select an acceptable course of action.

### 5.2.2 Anonymity mode

The SSA might also benefit onion routing services like Tor, used to preserve privacy and anonymity online. Currently Tor maintains its own browser, a fork of Firefox, to be used in conjunction with its services. This modified browser removes functionality that may inadvertently deanonymize users. The Tor client operates a local SOCKS proxy that encrypts and redirects traffic sent to it through Tor relays. One risk during Tor client use is that some identifying data may be leaked by other services on the machine that are not configured to use the SOCKS proxy. While it may be tempting to simply redirect all traffic from a machine through the SOCKS proxy, this is equally risky, due to the fact that not all applications are written with anonymity in mind, and may inadvertently leak identifying information through an exit node.

The SSA could be extended to support an "anonymity mode", which could be toggled by users. Using the elevated privileges of the kernel, this mode could block network activity for all applications except those specifically designed to operate in anonymous environments.

Applications could be developed specifically for this mode, perhaps requesting socket creation using a `SOCK_ANON` type, to be supported by the SSA. These features would allow applications to take advantage of onion routing without explicitly directing traffic to the SOCKS proxy, instead relying on the SSA for this redirection. In addition, applications not written to support anonymity would be restricted from using the network during the time the system is in anonymity mode. Naturally, developers of anonymity applications using `SOCK_ANON` would still have to take care that they do not inadvertently leak identifying information to the network. While the SSA could perform real-time traffic inspection to detect this, such an effort would be enormously complex, as different application protocols would have to be considered and data transmitted would have to be deeply vetted for privacy leaks.

### 5.2.3 Untrusted Environments

Much of the work presented has made the implicit assumption that an operating system is a trusted component. While this is necessarily the case in many environments, the emergence of cloud computing in recent years suggests use cases wherein the operating systems or hypervisors lie outside the realm of a user's trust. In these cases, application users and developers may be wary of unwanted tampering or eavesdropping by those who operate the remote servers on which their applications run. Fortunately, recent advancements in hardware such as Intel's Software Guard Extensions (SGX), AMD's Secure Encrypted Virtualization (SEV), allow applications to protect their sensitive data and code from higher-privileged entities, including hypervisors. Using these technologies, the SSA could protect key material even in remote cloud environments. In addition, userspace versions of the SSA could be constructed to be bundled with applications hosted in environments where even the operating system is not trusted.

### 5.2.4 CTAP Integration

The FIDO Alliance introduced the Client to Authenticator Protocol (CTAP), a generic application layer protocol for connecting an external authenticator device and a client platform. It defines transport-specific bindings for USB, NFC, and BlueTooth, but not WiFi as presented in our TLS client authentication work. Our authentication daemon could be modified to support CTAP, providing support for both TLS client authentication and authentication using other protocols. In addition, it may be possible to incorporate our WiFi support into the CTAP standard as 802.11 bindings. Such features could provide users with a single authentication solution for all their accounts.

### 5.2.5 TLS Client Authentication Infrastructure

Our work in TLS client authentication has numerous avenues for expansion. Notably absent from the presented work is an automated method for handling lost or stolen authentication devices. In this case the proper course of action is to revoke the certificate and issue a new one linked to the same user account. Doing this naturally requires a link between a user's account and some other identifying data apart from the lost key. Examples of such identifying data are email addresses and telephone numbers. For revocation and renewal operations, users could prove identity by providing evidence of their control over such identifiers (e.g, by receiving an email or SMS message and responding to a challenge contained therein). To increase usability, it would be preferable to have a mechanism that automated this process.

A standard protocol could be created that allows servers to automatically handle revocations and renewals using identity data stored within certificates initially issued. Ideally, the mobile application, acting on the user's behalf, could be extended to automatically use this protocol periodically to renew its certificates. Depending on the level of automation (i.e., lack of user interaction) such a protocol supports, this protocol could be used very frequently, minimizing the need for explicit revocation. This is similar in nature to the ACME protocol used by Let's Encrypt [45], which offers automated issuance of short-lived certificates.

While running a server that offers such automation would be beneficial for large organizations, smaller organizations or even individuals may wish to rely on a third party for this service. Thus another avenue for future work would be to create a certificate authority, analogous to Let's Encrypt, but for clients instead of servers. We tentatively term this effort Let's Authenticate. Let's Authenticate could operate using the aforementioned protocol and offer certificates to users with standard sets of identity attributes, requiring them to prove each one before certificate issuance. Server operators could then opt to trust the Let's Authenticate certificate authority instead of maintaining their own, and thus offload revocation and renewal responsibilities to another party. For example, the mobile app could be configured to periodically respond to challenge emails and SMS messages from a central Let's Authenticate server. This server would then transparently sign a certificate for the user, vouching for that user's ownership of her email address and telephone number. Upon registration to a website, the webserver could request a certificate from the Let's Authenticate authority, which the application could then provide.

An addition, the Let's Authenticate service could offer different tiers of certificates, each containing different amounts of identifying information. For example, Let's Encrypt could offer a tier of certificate that requires users to prove access to their mailing address, ownership of a driver's license, and other physically-based identity attributes, and includes these in the issued certificate. Such certificates would be useful in cases where more strict guarantees of user identity are needed, such as signing of home rental agreements online. The exploration of multiple tiers could also shed light on the amount of acceptable user security cost (in time and effort) with respect to different levels of security needs.

### 5.2.6 Furthering TLS Client Authentication Usability

The user study regarding our mobile application for TLS client authentication was merely preliminary, and many more can and should be performed to inform proper design for

widespread adoption. Here we list a set of additional user studies that can be performed to better inform application design and usability of our approach:

- **Different mediums:** BlueTooth, USB, NFC, and other communications mediums were not assessed in our user study, and could be explored to obtain data on user preferences and common pitfalls. In particular, it would be helpful to note the differences in user preference for BlueTooth and WiFi, and evaluate any difficulties users have with repeated pairings.
- **Long-term use:** Evaluating user experiences over a period of few days or weeks would be preferable to a short-lived laboratory experiment. A longer-term study would allow users to have more exposure to the interaction with the mobile app and better gauge their opinions of required interactions.
- **Real and multiple accounts:** Studies involving multiple types of accounts would provide insight into the types of accounts users want the mobile application to control. One issue with this approach is that participants would likely not be invested in fake accounts. A realistic configuration could be approximated if, with participant approval, coordinators installed a local proxy on home computers that translated password login forms to TLS client authentication requests. Users could privately supply this proxy with their account credentials during study setup, and the proxy could supply these credentials as needed during participant interaction with the mobile application and computer. While this would take some technical effort, the payoff would be that the mobile application could be evaluated in a setting where it is used with real accounts.
- **Simulated phone loss:** In conjunction with the renewal and revocation improvements mentioned in the previous subsection, a user study could be conducted that simulated loss of a private key. This could be part of a role play scenario in which a user loses his phone and is given a new one. Gracefully handling this scenario in a usable manner

is vital to adoption of this approach, especially considering the frequency with which users purchase new devices.

## 5.3 Enhancing the TLS Protocol

This dissertation focused on supporting TLS in the operating system without attempting to modify the protocol itself. However, this effort unveiled some shortcomings of the TLS protocol with respect to our goals. In this section we discuss two avenues for future exploration that modify the TLS protocol itself.

### 5.3.1 Native TLS Authentication Flexibility

While the works in this dissertation naturally solved many problems in TLS, some rigidity and inconsistencies in TLS itself created extra difficulties for our designs. One oddity in the TLS protocol is that its authentication process is not symmetric between the client and the server. During client authentication, the server provides the client a list of acceptable Distinguished Names of acceptable certificate authorities (or OID filters), allowing the client to select an appropriate certificate to present. During server authentication, the server is not afforded this opportunity, prohibiting servers from selecting an identity according to client desires. Such a mechanism would better facilitate authentication strengthening technologies, allowing clients to indicate that they would like the server to present a certificate from a more trustworthy CA, or a self-signed certificate used in Convergence.

Better than this, however, would be for the TLS protocol to not select a single method of authentication. X.509 certificates and the certificate authority system may not always be the the desired method of authentication, nor the only one desired, which is one motivation for TrustBase. Unfortunately, the use of X.509 certificates for both client and server authentication is part of the TLS specification. While some alternatives, such as pre-shared keys or TLS extensions like TLS-SRP, are available, a more flexible framework that enables clients and servers to agree upon any type of authentication is preferable. We

recommend that future versions of TLS incorporate authentication model flexibility into the handshake protocol. Such an effort could include a platform that supported authentications, like PAM or TrustBase's policy engine, to enable alternative client authentication strategies such as self-sovereign identity methods, Kerberos tickets, OPAQUE [46], etc.

### 5.3.2 Native TLS Identity Registration

Servers often obtain certificates from certificate authorities to prove their identity to clients. However, in practice servers typically authenticate client certificates by checking that they are issued by a certificate the server organization owns, rather than a third-party authority. This certificate registration is required to be done out of band, as the TLS protocol offers no built-in mechanism to request enrollment from a client. However, servers not using strong encryption typically enroll a client with a password at the application layer.

It would be preferable to have a procedure built directly into TLS, to provide TLS servers and clients a generic mechanism for enrollment. This could potentially be implemented as an optional protocol which follows client authentication requests in the TLS handshake protocol. Under such a protocol, the application layer would no longer be forced to implement security constructs, with encryption, authentication, and even registration handled by the security layer. This placement of registration in the security layer could lead to further SSA enhancements, especially if combined with the authentication model flexibility previously discussed.

## 5.4 Centralization Considerations

There are two principle risks involved in providing a centralized operating system service. First, the centralization aspect means that any flaw in the service becomes shared by all applications using that service. We have seen examples of such vulnerabilities in Apple's goto fail bug, and in OpenSSL's HeartBleed bug. While these vulnerabilities were severe, they were also quickly patched and a community effort was mounted to raise awareness and deploy

the fix. This illustrates that the drawback of centrality is also a feature. Having numerous independent organizations utilizing a service makes them a greater target for attackers, but also focuses their expertise and effort on a single implementation. An argument for centralized service is found within the long-standing security rule of "don't roll your own crypto".

The second risk concerns the kernel-level nature of the service. Kernel-level code is significantly more complex than traditional usermode code, and vulnerabilities within it can lead to full system compromise. In an effort to mitigate this concern, all of our solutions introduce minimal kernel code, in the form of loadable modules, and offload all major tasks (ASN.1 parsing, certificate validation encryption, etc.) to userspace daemon processes. The connection data within these processes can be further isolated through multiprocess options. While full-kernel implementations of our work would likely exhibit performance gains, great care must be taken to properly place all TLS functionality within the kernel. We recommend that those who pursue this course work closely with the security community to properly vet and design their implementations. On a related note, microkernels represent a useful tool to reconcile the risks of kernel services with their benefits. In such environments, a small kernel can run with the highest privilege, while other operating system services run sandboxed, but with higher privilege than applications. Due to this fact, microkernels may be a fruitful path forward for those seeking to expand the domain of operating system services in a robust manner.

## 5.5 Administrator-Developer Policy Conflicts

In moving to a model of OS-brokered network security, there may arise some tension between the security requirements of application developers and administrators. This may be caused by a developer that wants an application to connect to servers using legacy security protocols, or one that wants more advanced security than the system administrator allows. We consider this latter case for completeness, not expecting it to occur often in practice. However, it is conceivable that situations exist wherein an administrator values computational performance

to a degree that downgrades security to a level below what an application developer has guaranteed its clientele.

As a matter of personal freedom, we assert that the owner of a system should have the ultimate say in its security behaviors. However, we also respect the accountability that an application developer has to its users. The conservative recommendation to system designers is to only allow operations that lie within the intersection of application and administrator security requirements. This allows administrators to remain in control of their systems while guaranteeing concerned application developers that their applications will not be run under circumstances that do not match their policies. Another option, more in line with full administrator control, is to allow administrators to override applications' security requirements, even when this results in a downgrade. Under this option, it would be preferable to officially absolve applications developers of any resulting damage, akin to the breaking of a warranty seal on a physical product. How this absolution would work in practice, as well as any linkage to existing legal frameworks, is left to future exploration.

# Chapter 6

# Conclusion

The works described in this dissertation each introduce solutions to long-standing issues in Transport Layer Security. Each of them leverage the operating system to accomplish their stated goals. In doing so, they create a natural delineation of TLS responsibilities for users, application developers, and security experts. In this chapter, we discuss this delineation, the theoretical foundations that motivated the use of the operating system to solve problems, and the contributions of the associated works.

## 6.1 Brokered Security

Security is inherently both a restrictive and permissive effort. The restrictive facet of the field concerns itself with prohibiting access by unauthorized entities. Complementary to this is the permissive facet, which concerns itself with enabling authorized entities to operate unimpeded. Security flaws arise when there is a misalignment between these two facets, wherein unauthorized entities are permitted access, or wherein authorized entities are restricted from access. The prior set of flaws are addressed by conventional areas of security, such as encryption, which seek to provide confidentiality from unauthorized eavesdropping. The field of usable security can be considered an example of an effort to solve the latter flaws, as it seeks to enable users to perform security tasks with ease.

Sometimes conflict arises between these two facets, wherein an authorized entity has a clear need for access, but should be restricted from it due to the possible harm from misuse of that access. An example solution to this problem is the service of virtual memory, provided

by most operating systems. Applications have a clear need to access memory, as they cannot function otherwise. However, providing applications access to physical memory would be dangerous, as they could interfere with memory owned by the kernel or other applications, either maliciously or accidentally. Even benign and well-designed applications would incur an overwhelming amount of extra complexity in this case, as they would have to consider the possibility that any memory access may result in the storage or retrieval of values to and from foreign processes. A virtual memory service solves these problems elegantly, by acting as a broker that permits *indirect access* to physical memory. Under this service, applications are simultaneously *permitted* to access memory, but *restricted* from doing so in ways that harm the rest of the system. Furthermore, applications are provided with an abstraction of memory that gives the impression that they have sole access to (roughly) the entire address space of the machine, greatly reducing complexity for programmers. Finally, this brokerage by a privileged service enables easy configuration of memory access, allowing administrators to dictate various parameters, such as how much memory to provide each application.

Numerous aspects of contemporary TLS use are analogous to a world of memory use without virtual memory. Network applications have a clear need to use TLS to protect their data from unauthorized eavesdropping and tampering, but their misuse of TLS may result in vulnerabilities to various network attacks and put the system at risk. Without a brokering service, application developers are forced to implement TLS on their own, which has resulted in the abundance of problems discussed in Chapter 1, including the rigidity and improper use of certificate validation, the vulnerable implementation of TLS and its configuration, and the inability to use strong client authentication techniques. Related to each of these is the inability administrators and users have to custom tailor application security to their needs. These problem areas can thus be seen as merely symptomatic of a larger one – the absence of TLS as an operating system service.

The operating system is uniquely qualified to host security services. This is because it is the only entity within a computer system that can satisfy both facets of security. It has

complete control over restrictive policy, as it is the gatekeeper for all network interface access. It also has unique permissive powers, as it can deliver services to every application on a machine, enabling developers to rely on it for security, and can provide a unified configuration, enabling administrators to tailor security settings.

It is important to note that our use of the phrase "system administrators" refers to enterprise IT personnel, power users, and operating system vendors. This last group is important to consider, given that it is also responsible for continual shipment of security patches to end users, and typically have a well-established distribution channel for this purpose (e.g., Windows update, Aptitude).

We now outline the contributions of each work described in this dissertation, in the context of their ability to solve TLS problems through operating system restrictions and permissions.

## 6.2 TrustBase Contributions

TrustBase marks our first effort to bring TLS management into the operating system, targeting problems in server authentication. A primary goal of TrustBase is to support proper validation for all applications on a machine, regardless of what libraries they use or the language in which they are written. Through its use of a kernel-based traffic interceptor, TrustBase is able to accomplish this without the need to modify applications. The interceptor's ability to both monitor and modify traffic in transit provides it with permissive and restrictive abilities, respectively. Applications can rely on the operating system to validate certificates implicitly, relieving the burden on developers to perform these checks individually. Developers seeking more explicit use of this service can interface with the native validation API, obtaining validation assurance directly from the system. Servers bearing inadequate or falsified certificate chains are restricted from communicating with applications, as TrustBase terminates the connection before any data can be transmitted.

Related to the mistakes developers make in server authentication are the problems inherent in the certificate authority system. In an effort to alleviate the reliance on the certificate authority system, and even the reliance on any single validation system, TrustBase provides a pluggable policy engine. This allows TrustBase to provide additional permissiveness, enabling users to transparently gain the benefits of OSCP, CRLSets, certificate pinning, notary-based systems, and other solutions.

Finally, TrustBase provides permissions to administrators that enable them to tailor server authentication to their desired policies. These policies are enforced system wide, and no application can bypass them without explicit approval.

### 6.3 Secure Socket API Contributions

The difficulty developers have in adding TLS support to their network applications, and doing so securely, is evidence of a usability problem. Specifically, developer usability under traditional, userspace APIs is poor. The Secure Socket API addresses this problem by providing TLS functionality as an operating system service. This permits application developers to use TLS more easily, reducing the code required to use TLS from thousands to as little as one. The SSA does this for both server and client applications.

In an effort to increase usability, the SSA was designed to use only existing network function calls within the standard POSIX API. Because this interface is implemented by the kernel itself, the SSA offers the most minimal TLS API possible – built directly into existing kernel system calls, rather than into yet another userspace library for developers to learn. This permits even developers who are wholly unfamiliar with TLS to use its features, as they can invoke TLS functionality by using the standard networking API with which they are already familiar. In addition, the centralization of a TLS service allows security patches to be deployed system-wide, and applications using it do not have to be manually updated by developers. As a result, developers are no longer tasked with becoming security experts and can instead focus on features that make their applications unique.

The position of the SSA within the operating system allows it to extend its features to more than just applications written in C/C++. We demonstrated this through our minimal additions to the Go network library, and the PHP and Python interpreters, adding SSA support to each. Since all languages with networking support make use of networking system calls, directly or indirectly, the SSA easily supports any language.

The SSA, like TrustBase, provides permissions to administrators which enable fine-grained control. In this case, administrators can create system and per-application profiles that dictate the various TLS parameters that applications use. Among the profile options are TLS version selection, cipher suite selection, caching and session behavior, and certificate validation mechanism selection (which includes TrustBase). This eliminates the impossible burden on application developers to guess what the best security parameters are for their users, and places it on system designers and administrators, who are better-suited to make such decisions. Operating system vendors can provide secure-by-default settings, and update them in conjunction with standard patch cycles, and knowledgeable IT personnel can adapt these to their organization's needs.

Our SSA effort also features a restrictive component, through the use of a tool that coerces applications to use the SSA rather than the security libraries originally linked by the developer. In this sense, applications using other libraries can be viewed as entities attempting unauthorized access, as they are not subordinating their use of TLS to administrator control. Our tool prevents this, opting instead to emulate the behavior of the library and use the TLS functionality provided by the SSA, which does obey administrator preferences.

Another benefit of using the kernel for the SSA is that it resides in a different address space than the applications it supports. Thus, when an application is compromised, thereby becoming an unauthorized entity, the SSA acts as a restrictive force to block access to key material. Although an attacker may have compromised the application, he can only access TLS data using the SSA, which does not allow retrieval of key material. The isolation of the

SSA from applications also prohibits accidental or intentional modification of TLS parameter data by uncompromised applications.

## 6.4 Client Authentication Contributions

Our efforts to modernize TLS client authentication demonstrate the utility of having a centralized service for authentication tasks. This service permitted both server and client applications to easily implement strong client authentication, regardless of the physical medium used to transfer credential data to the system (e.g., BlueTooth, WiFi, etc.). Server developers can take advantage of TLS client authentication with as little as two lines of code, and the API mitigates prior security vulnerabilities that arose from mixing application and TLS logic. Under this system, client applications have no need to be modified to support client authentication, just as TrustBase allowed similar features to applications for server authentication.

These efforts further exemplified the power of separating the security and application layers by restricting the access applications had to key material and even other sensitive data contained within certificates. Responses from participants during the user study indicated that these extra protections are important to users, and these were made impossible without placing authentication tasks in the operating system. The concept of blinding applications from sensitive data leads to further questions regarding the types of data that should and should not be transmitted at various network layers, and we leave these questions for future work.

This separation also combats phishing in a natural way, as it divorces the authentication mechanism from the application requesting it. Even in the absence of password-replacement, such a mechanism can be leveraged to restrict credential input to predefined areas, such as a mobile authenticator's touchpad or an isolated OS-protected user interface. As discussed in the future work section of Chapter 4, this could be leveraged to implement TLS-SRP in a phishing-resistant manner.

Users were benefited by the modernization of TLS client authentication, too. With authentication under operating system control, authentication devices are provided a unified interface for authentication tasks, and users are given access to standard authentication controls such as logout. More importantly, users are given more security by not sharing their private keys with the system and given more privacy due to applications' inability to access certificate information.

## 6.5 Summary

We have seen many outstanding problems in TLS solved by the construction of a security layer, managed by the operating system. The services comprising this layer provide server authentication, transparent TLS implementation, and a platform for strong client authentication. By leveraging the centrality of the operating system and its privilege level, these services drastically reduce the security burden placed upon application developers, expand the flexibility of important security parameters, and empower system administrators. This approach to our stated problems is likely not unique to TLS, or even network security, and we encourage others to apply similar techniques to other areas of security. We hope that this work encourages system designers to adopt brokerage models that preserve the rights of owners to dictate how their devices operate, simplify and secure the use of platforms by developers, and leverage privilege to harmonize these goals.

## References


[1] Josh Aas. Let's encrypt root trusted by all major root programs. `https://letsencrypt.org/2018/08/06/trusted-by-all-major-root-programs.html`, 2018. Accessed: 6 August, 2018.

[2] Mansoor Alicherry and Angelos D Keromytis. Doublecheck: Multi-path verification against man-in-the-middle attacks. In *IEEE Symposium on Computers and Communications (ISCC)*, pages 557–563. IEEE, 2009.

[3] FIDO Alliance. FIDO alliance. `https://fidoalliance.org`, 2018. Accessed: 5 July, 2018.

[4] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Extracting certificates from live traffic: A near real-time SSL notary service. Technical report, TR-12-014, ICSI, 2012.

[5] Bernhard Amann, Matthias Vallentin, Seth Hall, and Robin Sommer. Revisiting SSL: A large-scale study of the Internet's most trusted protocol. Technical report, TR-12-015, ICSI, 2012.

[6] Leo St Amour and W Michael Petullo. Improving application security through TLS-library redesign. In *Security, Privacy, and Applied Cryptography Engineering (SPACE)*, pages 75–94. Springer, 2015.

[7] Apple Inc. What's new in iOS. `https://developer.apple.com/library/archive/releasenotes/General/WhatsNewIniOS/Articles/iOS9.html#/apple_ref/doc/uid/TP40016198-SW1`, 2017. Accessed: 01 June 2018.

[8] Dirk Balfanz and Ryan Hamilton. Transport layer security (TLS) channel IDs. Internet-Draft draft-balfanz-tls-channelid-01, IETF Secretariat, June 2013. URL `http://www.ietf.org/internet-drafts/draft-balfanz-tls-channelid-01.txt`. `http://www.ietf.org/internet-drafts/draft-balfanz-tls-channelid-01.txt`.

[9] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In *Advances in Cryptology*, pages 608–625. Springer, 2012.

[10] Adam Bates, Joe Pletcher, Tyler Nichols, Braden Hollembaek, Dave Tian, Kevin RB Butler, and Abdulrahman Alkhelaifi. Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security (CCS)*, pages 394–405, 2014.

[11] Henry Birge-Lee, Yixin Sun, Anne Edmundson, Jennifer Rexford, and Prateek Mittal. Bamboozling certificate authorities with BGP. In *USENIX Security Symposium*, pages 833–849, 2018.

[12] Joseph Bonneau, Cormac Herley, Paul C Van Oorschot, and Frank Stajano. The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. In *IEEE Symposium on Security and Privacy (SP)*, pages 553–567. IEEE, 2012.

[13] John Brooke. SUS: A quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[14] Chad Brubaker, Suman Jana, Baishakhi Ray, Sarfraz Khurshid, and Vitaly Shmatikov. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (SP)*, pages 114–129, 2014.

[15] Mauro Conti, Nicola Dragoni, and Sebastiano Gottardo. MITHYS: Mind the hand you shake - protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*, pages 65–81. Springer, 2013.

[16] D. Cooper, S. Santesson, S. Farrell, S. Boeyen, R. Housley, and W. Polk. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, RFC Editor, May 2008. URL `http://www.rfc-editor.org/rfc/rfc5280.txt`. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[17] Alexei Czeskis, Michael Dietz, Tadayoshi Kohno, Dan Wallach, and Dirk Balfanz. Strengthening user authentication through opportunistic cryptographic identity assertions. In *ACM Conference on Computer and Communications Security (CCS)*, pages 404–414. ACM, 2012.

[18] Anupam Das, Joseph Bonneau, Matthew Caesar, Nikita Borisov, and XiaoFeng Wang. The tangled web of password reuse. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[19] Xavier de Carné de Carnavalet and Mohammad Mannan. Killed by proxy: Analyzing client-end TLS interception software. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[20] T. Dierks and E. Rescorla. The transport layer security (TLS) protocol version 1.2. RFC 5246, RFC Editor, August 2008. URL `http://www.rfc-editor.org/rfc/rfc5246.txt`. `http://www.rfc-editor.org/rfc/rfc5246.txt`.

[21] Paul Ducklin. Anatomy of a password disaster Adobe's giant-sized cryptographic blunder. `https://nakedsecurity.sophos.com/2013/11/04/anatomy-of-a-password-disaster-adobes-giant-sized-cryptographic-blunder/`, 2013. Accessed: 1 July, 2018.

[22] Zakir Durumeric, James Kasten, Michael Bailey, and J Alex Halderman. Analysis of the HTTPS certificate ecosystem. In *ACM Internet Measurement Conference (IMC)*, pages 291–304, 2013.

[23] Zakir Durumeric, David Adrian, Ariana Mirian, James Kasten, Elie Bursztein, Nicolas Lidzborski, Kurt Thomas, Vijay Eranti, Michael Bailey, and J Alex Halderman. Neither snow nor rain nor MITM...: An empirical analysis of email delivery security. In *Internet Measurement Conference (IMC)*, pages 27–39, 2015.

[24] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J Alex Halderman, and Vern Paxson. The security impact of HTTPS interception. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2017.

[25] Peter Eckersley and Jesse Burns. An observatory for the SSLiverse. `http://www.eff.org/files/DefconSSLiverse.pdf`, 2010.

[26] Peter Eckersley and Jesse Burns. The (decentralized) SSL observatory (invited talk). In *USENIX Security Symposium*, 2011.

[27] Jake Edge. TLS in the kernel. `https://lwn.net/Articles/666509/`, 2015. Accessed: 15 December 2017.

[28] Electronic Frontier Foundation (EFF). The Sovereign Keys Project. `http:/www.eff.org/sovereign-keys/`, 2011.

[29] Kai Engert. MECAI - mutually endorsing CA infrastructure. `http://kuix.de/mecai`, 2012. Accessed: 21 September, 2016.

[30] Chris Evans and Chris Palmer. Certificate Pinning Extension for HSTS. Internet-Draft draft-evans-palmer-hsts-pinning-00, Internet Engineering Task Force, November 2011. URL https://tools.ietf.org/html/draft-evans-palmer-hsts-pinning-00. Work in Progress.

[31] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM Conference on Computer and Communications Security (CCS)*, pages 50–61. ACM, 2012.

[32] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. Rethinking SSL development in an appified world. In *ACM Conference on Computer and Communications Security (CCS)*, pages 49–60. ACM, 2013.

[33] Ian D. Foster, Jon Larson, Max Masich, Alex C Snoeren, Stefan Savage, and Kirill Levchenko. Security by any other name: On the effectiveness of provider based email security. In *ACM Conference on Computer and Communications Security (CCS)*, pages 450–464. ACM, 2015.

[34] OpenSSL Software Foundation. 1.0.2 manpages. https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_new.html, 2015. Accessed: 15 December 2017.

[35] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: Validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)*, pages 38–49. ACM, 2012.

[36] Dan Goodin. Google Chrome will banish Chinese certificate authority for breach of trust. http://arstechnica.com/security/2015/04/google-chrome-will-banish-chinese-certificate-authority-for-breach-of-trust/, 2015.

[37] Dani Grant. Introducing TLS with client authentication. https://blog.cloudflare.com/introducing-tls-client-auth/, 2018. Accessed: 5 November, 2018.

[38] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLint. In *IEEE Symposium on Security and Privacy (SP)*, pages 519–534. IEEE, 2015.

[39] P. Hoffman and J. Schlyter. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. Internet Requests for Comments,

August 2012. ISSN 2070-1721. URL http://www.rfc-editor.org/rfc/rfc6698.txt. http://www.rfc-editor.org/rfc/rfc6698.txt.

[40] Matthew Holt. Caddy. https://caddyserver.com/, 2015. Accessed: 15 April 2018.

[41] Ralph Holz, Thomas Riedmaier, Nils Kammenhuber, and Georg Carle. X.509 forensics: Detecting and localising the SSL/TLS men-in-the-middle. In *European Symposium on Research in Computer Security (ESORICS)*, pages 217–234. Springer, 2012.

[42] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2016.

[43] Russ Housley and Ralph Droms. TLS 1.3 option for negotiation of visibility in the datacenter. Internet-Draft draft-rhrd-tls-tls13-visibility-01, IETF Secretariat, March 2018. URL http://www.ietf.org/internet-drafts/draft-rhrd-tls-tls13-visibility-01.txt. http://www.ietf.org/internet-drafts/draft-rhrd-tls-tls13-visibility-01.txt.

[44] Lin-Shung Huang, Alex Rice, Erling Ellingsen, and Collin Jackson. Analyzing forged SSL certificates in the wild. In *IEEE Symposium on Security and Privacy (SP)*, pages 83–97, 2014.

[45] Internet Security Research Group (ISRG). Let's encrypt. https://letsencrypt.org/, 2018. Accessed: 5 September, 2018.

[46] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. OPAQUE: An asymmetric PAKE protocol secure against pre-computation attacks. In *Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2018.

[47] Gregg Keizer. Hackers spied on 300,000 Iranians using fake Google certificate. http://www.computerworld.com/article/2510951/cybercrime-hacking/hackers-spied-on-300-000-iranians-using-fake-google-certificate.html, 2011. Accessed: 27 October, 2015.

[48] Tiffany Hyun-Jin Kim, Lin-Shung Huang, Adrian Perring, Collin Jackson, and Virgil Gligor. Accountable key infrastructure (AKI): A proposal for a public-key validation infrastructure. In *International Conference on World Wide Web (WWW)*, pages 679–690, 2013.

[49] B. Laurie, A. Langley, and E. Kasper. Certificate transparency. RFC 6962, RFC Editor, June 2013.

[50] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The art of memory forensics: Detecting malware and threats in Windows, Linux, and Mac memory*. John Wiley & Sons, 2014.

[51] Yabing Liu, Will Tome, Liang Zhang, David Choffnes, Dave Levin, Bruce Maggs, Alan Mislove, Aaron Schulman, and Christo Wilson. An end-to-end measurement of certificate revocation in the web's PKI. In *ACM Internet Measurement Conference (IMC)*, pages 183–196, 2015.

[52] Kai-Uwe Loser and Martin Degeling. Security and privacy as hygiene factors of developer behavior in small and agile teams. In *IFIP International Conference on Human Choice and Computers (HCC)*, pages 255–265. Springer, 2014.

[53] Michael Maass, Adam Sales, Benjamin Chung, and Joshua Sunshine. A systematic analysis of the science of sandboxing. *PeerJ Computer Science*, 2:e43, 2016.

[54] Mohammad Mannan and Paul C Van Oorschot. Using a personal device to strengthen password authentication from an untrusted computer. In *Financial Cryptography and Data Security*, pages 88–103. Springer, 2007.

[55] Moxie Marlinspike. SSL and the future of authenticity. *Black Hat USA*, 2011.

[56] Moxie Marlinspike and Trevor Perrin. Trust assertions for certificate keys. `http://tack.io/`, 2013. Accessed: 1 May, 2017.

[57] Nikos Mavrogiannopoulos. Fedora system-wide crypto policy. `http://fedoraproject.org/wiki/Changes/CryptoPolicy`, 2015. Accessed: 15 December 2017.

[58] Nikos Mavrogiannopoulos, Miloslav Trmač, and Bart Preneel. A Linux kernel cryptographic framework: decoupling cryptographic keys from applications. In *ACM Symposium on Applied Computing*, pages 1435–1442. ACM, 2012.

[59] Jonathan M McCune, Adrian Perrig, and Michael K Reiter. Bump in the ether: A framework for securing sensitive user input. In *USENIX Annual Technical Conference*, pages 17–17, 2006.

[60] David Meyer. Nokia: Yes, we decrypt your HTTPS data, but don't worry about it. `http://gigaom.com/2013/01/10/`

nokia-yes-we-decrypt-your-https-data-but-dont-worry-about-it/, 2013. Accessed 10 January, 2013.

[61] Erik Nygren. Reaching toward universal TLS SNI. https://blogs.akamai.com/2017/03/reaching-toward-universal-tls-sni.html, 2017. Accessed: 21 June, 2017.

[62] Devon O'Brien, Ryan Sleevi, and Andrew Whalley. Chrome's plan to distrust Symantec certificates. https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html, 2017. Accessed: 1 July, 2018.

[63] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. It's the psychology stupid: How heuristics explain software vulnerabilities and how priming can illuminate developer's blind spots. In *Annual Computer Security Applications Conference (ACSAC)*, pages 296–305. ACM, 2014.

[64] Mark O'Neill. The state of man-in-the-middle TLS proxies: Prevalence and user attitudes. In *BYU ScholarsArchive*. Brigham Young University, 2016.

[65] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. TLS proxies: Friend or foe? In *ACM Internet Measurement Conference (IMC)*, pages 551–557, 2016.

[66] Mark O'Neill, Scott Heidbrink, Scott Ruoti, Jordan Whitehead, Dan Bunker, Luke Dickinson, Travis Hendershot, Joshua Reynolds, Kent Seamons, and Daniel Zappala. TrustBase: An architecture to repair and strengthen certificate-based authentication. In *USENIX Security Symposium*, pages 609–624, 2017.

[67] Mark O'Neill, Scott Heidbrink, Jordan Whitehead, Tanner Perdue, Luke Dickinson, Torstein Collett, Nick Bonner, Kent Seamons, and Daniel Zappala. The secure socket API: TLS as an operating system service. In *USENIX Security Symposium*, pages 799–816, 2018.

[68] Lucky Onwuzurike and Emiliano De Cristofaro. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In *ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec)*, pages 1–6. ACM, 2015.

[69] OpenBSD. LibreSSL. https://www.libressl.org/, 2014. Accessed: 12 May 2017.

[70] Bryan Parno, Cynthia Kuo, and Adrian Perrig. Phoolproof phishing prevention. In *Financial Cryptography and Data Security*, pages 1–19. Springer, 2006.

[71] Arnis Parsovs. Practical issues with TLS client certificate authentication. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[72] Lindsey Poole and Vivek S Pai. ConfiDNS: Leveraging scale and history to detect compromise. In *USENIX Annual Technical Conference*, pages 99–112, 2008.

[73] The Chromium Projects. CRLSets. `https://dev.chromium.org/Home/chromium-security/crlsets`, 2012. Accessed: 23 May 2018.

[74] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *USENIX Security Symposium*, pages 16–16, 2003.

[75] Steve Ragan. Adobe confirms stolen passwords were encrypted, not hashed. `https://csoonline.com/article/2134124/network-security/adobe-confirms-stolen-passwords-were-encrypted-not-hashed.html`, 2013. Accessed: 1 July, 2018.

[76] David Recordon and Drummond Reed. OpenID 2.0: A platform for user-centric identity management. In *ACM workshop on Digital Identity Management (DIM)*, pages 11–16. ACM, 2006.

[77] Ivan Ristić. Bulletproof SSL and TLS. *Feisty Duck*, 2014.

[78] Daniel Roethlisberger. Sslsplit. `https://www.roe.ch/SSLsplit`, 2009. Accessed: 11 July, 2015.

[79] Scott Ruoti, Mark O'Neill, Daniel Zappala, and Kent Seamons. User attitudes toward the inspection of encrypted traffic. In *Symposium on Usable Privacy and Security (SOUPS)*, pages 131–146, 2016.

[80] Mark Ryan. Enhanced certificate transparency and end-to-end encrypted mail. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[81] S. Santesson, M. Myers, R. Ankney, A. Malpani, S. Galperin, and C. Adams. X.509 internet public key infrastructure online certificate status protocol - OCSP. RFC 6960, RFC Editor, June 2013. URL `http://www.rfc-editor.org/rfc/rfc6960.txt`. `http://www.rfc-editor.org/rfc/rfc6960.txt`.

[82] Cory Scott. Protecting our members. `https://blog.linkedin.com/2016/05/18/protecting-our-members`, 2016. Accessed: 1 July, 2018.

[83] Vicente Silviera. An update on LinkedIn member passwords compromised. `https://blog.linkedin.com/2012/06/06/linkedin-member-passwords-compromised`, 2012. Accessed: 1 July, 2018.

[84] Christopher Soghoian and Sid Stamm. Certified lies: Detecting and defeating government interception attacks against SSL (short paper). In *Financial Cryptography and Data Security*, pages 250–259. Springer, 2012.

[85] David Sounthiraraj, Justin Sahs, Garret Greenwood, Zhiqiang Lin, and Latifur Khan. SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2014.

[86] D. Taylor, T. Wu, N. Mavrogiannopoulos, and T. Perrin. Using the secure remote password (SRP) protocol for TLS authentication. RFC 5054, RFC Editor, November 2007.

[87] W3C. Web authentication: An API for accessing public key credentials level 1. `https://w3.org/TR/2018/CR-webauthn-20180320/`, 2018. Accessed: 5 July, 2018.

[88] Matthias Wachs, Quirin Scheitle, and Georg Carle. Push away your privacy: Precise user tracking based on TLS client certificate authentication. In *Network Traffic Measurement and Analysis Conference (TMA)*, pages 1–9. IEEE, 2017.

[89] Dan Wendlandt, David G. Andersen, and Adrian Perrig. Perspectives: Improving SSH-style host authentication with multi-path probing. In *USENIX Annual Technical Conference*, pages 321–334, 2008.

[90] Alma Whitten and J Doug Tygar. Why Johnny can't encrypt: A usability evaluation of PGP 5.0. In *USENIX Security Symposium*, pages 169–184, 1999.

[91] Ubuntu Wiki. AppArmor profiles. `https://wiki.ubuntu.com/SecurityTeam/KnowledgeBase/AppArmorProfiles`, 2018. Accessed: 23 May 2018.

[92] Thomas YC Woo, Raghuram Bindignavle, Shaowen Su, and Simon S Lam. SNP: An interface for secure network programming. In *USENIX Summer Technical Conference*, pages 45–58, 1994.

[93] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy (SP)*, pages 590–604. IEEE, 2014.

[94] Yubico. Yubico — YubiKey strong two factor authentication for business and individual use. `https://yubico.com`, 2018. Accessed: 5 July, 2018.

[95] Hubert Zimmermann. OSI reference model–the ISO model of architecture for open systems interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

# Appendix A

# Client Authentication User Study Materials

# Participant Guide

For this study, we would like you to play the role of a shopper who wants to purchase a pair of shoes from an online retailer. You will use our phone as if it were your own, and our computer as if it were your own (the name of the computer has been set to reflect this).

At some point, you will need to register for an account or make a purchase. Use the smartphone we have provided and the Securely app for these steps. The Securely app has also been preloaded with credit card and shipping information that you will need for this task.

Please do the following:

- ☐ Set up the "Securely" app on the phone by opening the app and following its instructions.
- ☐ Register an account with paymore.com and log in to it.
- ☐ Choose any shoes that you like and add them to your shopping cart. You can buy whatever you'd like - the money used is not real.
- ☐ When you are ready, start the checkout process to purchase the shoes.
- ☐ Once you have completed the purchase, the study coordinator will guide you to the next task.

## End-of-Task Survey

*This survey will be given to all participants.*

For this portion of the study, we would like some feedback from you about using the Securely app to make accounts, log in, and make purchases.

*Page Break*

There are two aspects of the app that we have not shown you, but may be relevant to your answers in this survey:

1) Securely only needs to be set up once. You do not have to re-enter your information for each account you create (but you can if you'd like).

2) You only need to scan the QR code (barcode) once per computer. For the same computer tomorrow it would not ask you to scan the QR code. Scanning this code secures the link between your phone and the computer.

*Page Break*

**Please answer the following questions about the app (Securely). Try to give your immediate reaction to each statement without pausing to think for a long time. Mark the middle column if you don't have a response to a particular statement.**

1. I think that I would like to use the Securely app frequently.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree
    - Somewhat agree
    - Strongly agree
2. I found the Securely app unnecessarily complex.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree
    - Somewhat agree
    - Strongly agree
3. I thought the Securely app was easy to use.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree
    - Somewhat agree
    - Strongly agree
4. I think that I would need the support of a technical person to be able to use the Securely app.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree

- Somewhat agree
- Strongly agree

5. I found the various functions in the Securely app were well integrated.
   - Strongly disagree
   - Somewhat disagree
   - Neither agree nor disagree
   - Somewhat agree
   - Strongly agree
6. I thought there was too much inconsistency in the Securely app.
   - Strongly disagree
   - Somewhat disagree
   - Neither agree nor disagree
   - Somewhat agree
   - Strongly agree
7. I would imagine that most people would learn to use the Securely app very quickly.
   - Strongly disagree
   - Somewhat disagree
   - Neither agree nor disagree
   - Somewhat agree
   - Strongly agree
8. I found the Securely app very cumbersome to use.
   - Strongly disagree
   - Somewhat disagree
   - Neither agree nor disagree
   - Somewhat agree
   - Strongly agree
9. I felt very confident using the Securely app.
   - Strongly disagree
   - Somewhat disagree
   - Neither agree nor disagree
   - Somewhat agree
   - Strongly agree
10. I needed to learn a lot of things before I could get going with the Securely app.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree

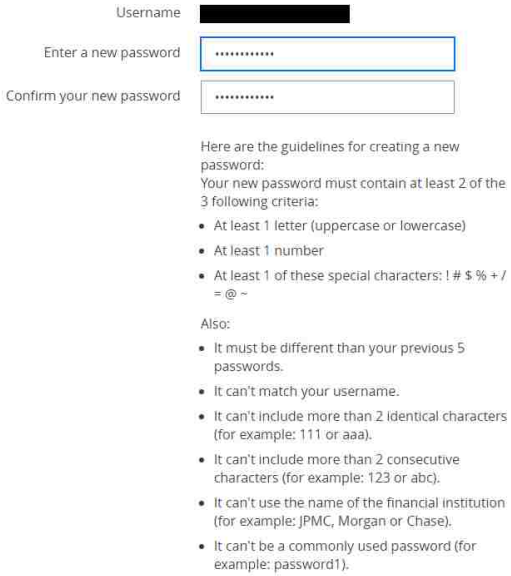
- Somewhat agree
- Strongly agree

11. What did you like about the Securely app?
(*Free response*)

12. What did you dislike about the Securely app?
(*Free response*)

Please take a moment to think about the last time you used passwords for an online account. Consider the steps needed for registration (with password creation) and logging in. As a refresher, here are some screenshots for two typical password registrations:


(a) Amazon registration page




(b) Chase Bank registration page

13. I would prefer to use the Securely app instead of passwords.
    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree
    - Somewhat agree
    - Strongly agree
14. I would prefer to use the Securely app instead of passwords for
    - None of my accounts.
    - Some of my accounts.
    - Half of my accounts.
    - Most of my accounts.
    - All of my accounts.

15. What types of accounts would you like to use the Securely app with (if any)?
(*Free response*)

16. What types of accounts would you NOT like to use the Securely app with (if any)?
(*Free response*)

17. The Securely app uses encryption and digital credentials instead of passwords. Because of this, no password ever needs to be sent to the website. As a result, you do not need to remember passwords, hackers cannot guess your password, and your password cannot be stolen from a website. In addition, the digital credentials use strong encryption that is extremely hard for hackers to break, compared to passwords.

    How important is this feature to you?

    - Not important
    - Somewhat unimportant
    - Neither unimportant or important
    - Somewhat important
    - Very important

18. The Securely app can also be used to choose what information you send to websites for registration. For example, you could register an account with only a username (and no real name, email address, etc.), or register an account with a fake name or separate email account for websites where you don't care to be identified. You could even have a separate, anonymous credentials for each website.

    How important is this feature to you?

    - Not important
    - Somewhat unimportant
    - Neither unimportant or important
    - Somewhat important
    - Very important

19. As demonstrated by the secure checkout process during the study, the Securely app can also be used to protect more than login credentials, such as credit cards, shipping and email addresses, etc.). When information is used this way, even the application on your computer (like your browser) cannot see it, which protects it from malware.

    How important is this feature to you?

    - Not important
    - Somewhat unimportant
    - Neither unimportant or important
    - Somewhat important
    - Very important

20. When you log in or use a credit card with the Securely app, it makes sure that you have an existing account with the site you are trying to visit. This prevents you from being "phished" (being tricked into logging in to an imposter site).

    How important is this feature to you?

    - Not important
    - Somewhat unimportant

- Neither unimportant or important
- Somewhat important
- Very important

21. The Securely app requires you to have your phone with you if you want to be able to log in to your accounts with the credentials stored in your phone. Your phone does need service or Internet access, but it does need to connect to the laptop or desktop computer using WiFi or BlueTooth.

    How much of a concern is this for you?

    - Not at all concerned
    - Slightly concerned
    - Somewhat concerned
    - Moderately concerned
    - Extremely concerned

22. If you lose your phone, you will not be able to log in to your accounts, since the Securely app stores your credentials in your phone. If you have a pin or pattern set for logging into your phone, your credentials can still be protected from unauthorized use. Credentials cannot be copied from the phone, either. We will eventually provide a credential reset system to deal with lost or stolen phones.

    How much of a concern is this for you?

    - Not at all concerned
    - Slightly concerned
    - Somewhat concerned
    - Moderately concerned
    - Extremely concerned

23. Knowing about these features, have your preferences to using the Securely app versus passwords changed?

    - Yes
    - No

    *If the participant selects no, the following three questions will not appear*

24. With this new information, I would prefer to use the Securely app instead of passwords.

    - Strongly disagree
    - Somewhat disagree
    - Neither agree nor disagree
    - Somewhat agree
    - Strongly agree

25. With this new information, I would prefer to use the Securely app instead of passwords for

    - None of my accounts.
    - Some of my accounts.
    - Half of my accounts.
    - Most of my accounts.

- All of my accounts.

26. What information caused you to change your mind and why?
(*Free response*)

27. What is your gender?
    - Male
    - Female
    - I prefer not to answer

28. What is your age?
    - 18-24 years old
    - 25-34 years old
    - 35-44 years old
    - 45-54 years old
    - 55 years or older
    - I prefer not to answer

29. What is the highest degree or level of school you have completed?
    - Some school, no high school diploma
    - High school graduate, diploma or the equivalent (for example: GED)
    - Some college or university credit, no degree
    - College or university degree
    - Post-Secondary Education
    - I prefer not to answer

30. What is your occupation or major?
(*Free response*)

31. How would you rate your level of computer expertise? Use the following descriptions as a guide to answering this question:

    **Beginner** (1 and 2 on the scale): Able to use a mouse and keyboard, create a simple document, send and receive e-mail, and/or access web pages.
    **Intermediate** (3 on the scale): Able to format documents using styles or templates, use spreadsheets for custom calculations and charts, and/or use graphics/web publishing.
    **Expert** (4 and 5 on the scale): Able to use macros in programs to speed tasks, configure operating system features, create a program using a programming language, and/or develop a database.

    - 1 (Beginner)
    - 2
    - 3 (Intermediate)
    - 4
    - 5 (Expert)