The Secure Socket API: TLS as an Operating System Service

Mark O'Neill, Scott Heidbrink, Jordan Whitehead, Tanner Perdue, Luke Dickinson, Torstein Collett, Nick Bonner, Kent Seamons, and Daniel Zappala, *Brigham Young University*

https://www.usenix.org/conference/usenixsecurity18/presentation/oneill

This paper is included in the Proceedings of the 27th USENIX Security Symposium.

August 15–17, 2018 • Baltimore, MD, USA

ISBN 978-1-939133-04-5

The Secure Socket API: TLS as an Operating System Service

Mark O'Neill Scott Heidbrink Jordan Whitehead Tanner Perdue
Luke Dickinson Torstein Collett Nick Bonner
Kent Seamons Daniel Zappala
Brigham Young University
mto@byu.edu, sheidbri@byu.edu, jaw@byu.edu, tanner_perdue@byu.edu
luke@isrl.byu.edu, torstein.collett@byu.edu, jbonner6@byu.edu
seamons@cs.byu.edu, zappala@cs.byu.edu

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.

1 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 [11].

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

Significant efforts to catalog developer mistakes and the complexities of modern security APIs have been published in recent years [8, 12, 23, 4, 19]. As a result, projects have emerged that reduce the size of security APIs [20], enhance library security [1], and perform certificate validation checks on behalf of vulnerable applications [3, 18, 9, 5]. 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 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.

2 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 [11, 12] 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 [4, 11, 8].

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 [15]. 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 [17]. Thus we believe establishing operating system and administrator control of TLS and related security policies is in line with precedent and best practice.

3 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.

4 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 the symbols into the groups shown in Table 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 [26], 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.

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 1: Breakdown of OpenSSL's `libssl` symbols.

4.1 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 [10]. 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.

4.2 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 hardcoded 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.

4.3 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.

4.4 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.

4.5 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 [11, 4, 8]. 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 [18, 3]. 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 [22], CRLs [6], 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.

4.6 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.

4.7 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.

4.8 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.

5 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.

5.1 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 `IPPROTP_TLS` as well. Table 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 1. Both the client and the server create a socket with the `IPPROTO_TLS` 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.

5.2 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).

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 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\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 response */
printf("Received:\n%s", http_response);
return 0;
```

(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 1: Code examples for applications using the SSA.

- **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.

5.2.1 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 [24], 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.

5.2.2 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 [18]. Under 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

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: Sample of socket options at the IPPROTO_TLS level

consisting of combinations of standard validation, OCSP checking [22], Google CRLset checking [21], certificate pinning, and DANE [13]. Additional validation mechanisms not listed can also be used, such as notary-based validation, through the TrustBase plugin API.

5.3 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. 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 ex-

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 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

tension. 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.

The options listed in Table 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.

5.4 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 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.

5.5 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 6.

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 [14], 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.

5.6 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_0RTT` socket option.

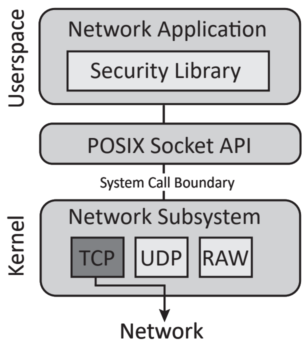

Figure 2: Data flow for traditional TLS library by network applications. The application shown is using TCP.

6 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 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, illustrates how our implementation of the SSA compares to this normal usage. We split our SSA implementation into two 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

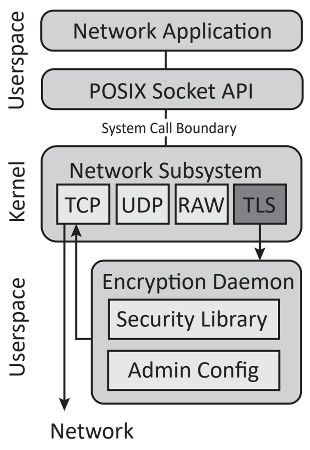

Figure 3: Data flow for SSA usage by network applications. The application shown is using the TLS (which uses TCP internally for connection-based `SOCK_STREAM` sockets).

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 5, 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.

6.1 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 [18], 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.

6.2 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 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 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.

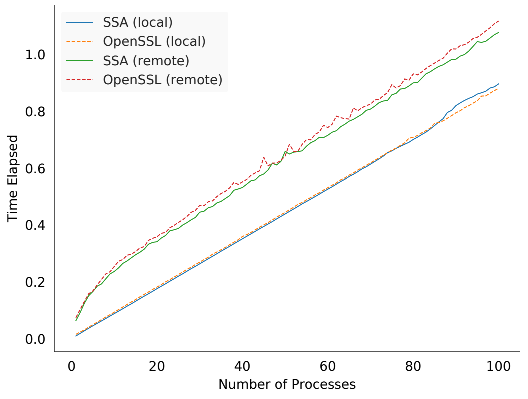

Figure 4: Time to transfer 1MB over LAN and WAN via HTTPS for applications using OpenSSL and the SSA, with varying numbers of simultaneous processes.

7 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. [3] 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.

8 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.

8.1 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 [16]. 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. [3], 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.

8.2 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 [16].

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.

8.3 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 [24]. 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 [15]. 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.

8.4 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 [7] 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.

8.5 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.

9 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.

10 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 2. Here we outline related work that aims at simplifying and securing TLS libraries, and improving administrator control.

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 [1]. 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 [20] 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) [25] 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.

Administrator control over TLS: Fahl et al. [9], MITHYS [5] and two other solutions, TrustBase [18] and CertShim [3], 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 [13], 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 [2] (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.

11 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.

References

[1] AMOUR, L. S., AND PETULLO, W. M. Improving application security through TLS-library redesign. In *Security, Privacy, and Applied Cryptography Engineering (SPACE)*. Springer, 2015, pp. 75–94.

[2] 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`. Accessed: 01 June 2018.

[3] BATES, A., PLETCHER, J., NICHOLS, T., HOLLEMBAEK, B., TIAN, D., BUTLER, K. R., AND ALKHELAIFI, A. Securing SSL certificate verification through dynamic linking. In *ACM Conference on Computer and Communications Security (CCS)* (2014), pp. 394–405.

[4] BRUBAKER, C., JANA, S., RAY, B., KHURSHID, S., AND SHMATIKOV, V. Using frankencerts for automated adversarial testing of certificate validation in SSL/TLS implementations. In *IEEE Symposium on Security and Privacy (SP)* (2014), IEEE, pp. 114–129.

[5] CONTI, M., DRAGONI, N., AND GOTTARDO, S. MITHYS: Mind the hand you shake-protecting mobile devices from SSL usage vulnerabilities. In *Security and Trust Management*. Springer, 2013, pp. 65–81.

[6] COOPER, D., SANTESSON, S., FARRELL, S., BOEYEN, S., HOUSLEY, R., AND POLK, W. Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile. RFC 5280, RFC Editor, May 2008. `http://www.rfc-editor.org/rfc/rfc5280.txt`.

[7] EDGE, J. TLS in the kernel. `https://lwn.net/Articles/666509/`. Accessed: 15 December 2017.

[8] FAHL, S., HARBACH, M., MUDERS, T., BAUMGÄRTNER, L., FREISLEBEN, B., AND SMITH, M. Why Eve and Mallory love Android: An analysis of Android SSL (in) security. In *ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 50–61.

[9] FAHL, S., HARBACH, M., PERL, H., KOETTER, M., AND SMITH, M. Rethinking SSL development in an appified world. In *ACM Conference on Computer and Communications Security (CCS)* (2013), ACM, pp. 49–60.

[10] FOUNDATION, O. S. 1.0.2 manpages. `https://www.openssl.org/docs/man1.0.2/ssl/SSL_CTX_new.html`. Accessed: 15 December 2017.

[11] GEORGIEV, M., IYENGAR, S., JANA, S., ANUBHAI, R., BONEH, D., AND SHMATIKOV, V. The most dangerous code in the world: validating SSL certificates in non-browser software. In *ACM Conference on Computer and Communications Security (CCS)* (2012), ACM, pp. 38–49.

[12] He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V., Yang, R., and Zhang, Z. Vetting SSL usage in applications with SSLint. In *IEEE Symposium on Security and Privacy (SP)* (2015), IEEE, pp. 519–534.

[13] Hoffman, P., and Schlyter, J. The DNS-based authentication of named entities (DANE) transport layer security (TLS) protocol: TLSA. Internet Requests for Comments, August 2012. `http://www.rfc-editor.org/rfc/rfc6698.txt`.

[14] Holt, M. Caddy. `https://caddyserver.com/`. Accessed: 15 April 2018.

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

[16] Mavrogiannopoulos, N., Trmač, M., and Preneel, B. A Linux kernel cryptographic framework: decoupling cryptographic keys from applications. In *ACM Symposium on Applied Computing* (2012), ACM, pp. 1435–1442.

[17] Oliveira, D., Rosenthal, M., Morin, N., Yeh, K.-C., Cappos, J., and Zhuang, Y. 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)* (2014), ACM, pp. 296–305.

[18] O'Neill, M., Heidbrink, S., Ruoti, S., Whitehead, J., Bunker, D., Dickinson, L., Hendershot, T., Reynolds, J., Seamons, K., and Zappala, D. TrustBase: An architecture to repair and strengthen certificate-based authentication. In *USENIX Security Symposium* (2017).

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

[20] OpenBSD. LibreSSL. `https://www.libressl.org/`. Accessed: 12 May 2017.

[21] Projects, T. C. CRLSets. `https://dev.chromium.org/Home/chromium-security/crlsets`. Accessed: 23 May 2018.

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

[23] Sounthiraraj, D., Sahs, J., Greenwood, G., Lin, Z., and Khan, L. SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps. In *Network and Distributed System Security Symposium (NDSS)* (2014).

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

[25] Woo, T. Y., Bindignavle, R., Su, S., and Lam, S. S. SNP: An interface for secure network programming. In *USENIX Summer Technical Conference* (1994), pp. 45–58.

[26] Yamaguchi, F., Golde, N., Arp, D., and Rieck, K. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy (SP)* (2014), IEEE, pp. 590–604.