Poster: The Secure Socket API

Mark O'Neill
Computer Science
Brigham Young University
Provo, USA
mto@byu.edu

Scott Heidbrink
Computer Science
Brigham Young University
Provo, USA
sheidbri@byu.edu

Kent Seamons
Computer Science
Brigham Young University
Provo, USA
seamons@cs.byu.edu

Daniel Zappala
Computer Science
Brigham Young University
Provo, USA
zappala@cs.byu.edu

I. Introduction

Transport Layer Security is the protocol most-responsible for encryption on the Internet today. Unfortunately, popular TLS security libraries, such as OpenSSL and GnuTLS, while feature-rich and widely-used, have long been plagued by programmer misuse, leading to security flaws. The complexity and design of these libraries can make them hard to use correctly for application developers and even security experts.

In this work we present the Secure Socket API (SSA), a TLS API 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. The SSA enables developers to quickly build TLS support into their applications and administrators to easily control how applications use TLS on their machines. This reduces application code by thousands of lines, and reduces security APIs from over 500 functions to a mere dozen. 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.

II. 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 library alone exports 504 functions and macros for use by TLS-implementing applications. This and other TLS APIs have been criticized for their complexity [3], [4] and, anecdotally, our own explorations find many functions within `libssl` that have non-intuitive semantics, confusing names, or little-to-no use in applications. Another body of work has cataloged developer mistakes when using these libraries to validate certificates, resulting in man-in-the-middle vulnerabilities [1]–[3].

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.

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

III. Basic Design

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 utilize the selected protocol transparently.

We have developed a prototype SSA as a loadable Linux kernel module. This module extends the operating system networking interface, adding the `IPPROTO_TLS` protocol option to `socket`. When a developer creates a TLS socket, subsequent calls to POSIX socket functions such as `connect`, `send`, and `recv` would then properly utilize the TLS handshake, encrypt and transmit data, and receive and decrypt data respectively, based on the TLS protocol.

IV. Administrator Control

The SSA is responsible for automatic selection of TLS versions, cipher suites, and extensions. It also performs automatic session management and automatic validation of certificates. These behaviors are subject to a system configuration policy with secure defaults.

Administrators can customize the behavior of the SSA through a protected configuration file, controlling TLS version and cipher suite selection, certificate validation strategies, TLS extensions, etc. Settings are applied to all TLS connections made with the SSA. These can be further tailored to individual applications through the creation of additional profiles, which contain settings for specific applications.

V. Developer Customization

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

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 I: 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 limited set of principal network functions. In accordance with this standard, the SSA adds a few socket options for `IPPROTO_TLS`. This set includes functionality for developers to specify remote hostnames, local certificate chains and private keys, custom certificate validation, session TTLs, etc. Administrators set policy, while developers can choose to further restrict an application, such as choosing a particular cipher suite out of the configured options. Developers can increase security but cannot decrease it.

VI. Porting Applications

To obtain metrics on porting applications to use the SSA, we modified the source code of four network programs to use the SSA. Two of these already utilized OpenSSL for their TLS functionality, and two were not built to utilize TLS at all. The lines of code modified, removed, and the time taken to accomplish the conversion are shown in Table I. The modification of `wget` and `lighttpd` were performed by programmers with no prior experience with the source code or OpenSSL, but who had a working knowledge of C and POSIX sockets. Most of the time spent was used to become familiar with the source code and remove OpenSSL calls.

These results 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.

VII. Porting Languages

One of the benefits of using the POSIX socket API as the basis for the SSA is that it makes it easy to provide support for the SSA in a variety of languages, since the POSIX API is often implemented at the system call layer. 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.

To illustrate this, we have added SSA support to three additional languages beyond C/C++: Go, Python, and PHP. In each case the modifications needed were light: Python and PHP merely required additional definitions of the SSA constants, and Go required these plus some simple wrappers (2-3 lines of code each) to be created for its `setsockopt`/`getsockopt` interface. With the changes to the Go standard library, we successfully modified the popular Caddy webserver to use the SSA with only the modification of a single line of its code.

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.

VIII. Discussion

The SSA itself and the architecture of our prototype both have compelling 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 in a familiar API. This simplified TLS interface allows developers to focus on unique application logic, rather than spending time implementing standard network security with complex APIs.

Because our SSA design moves all TLS functionality to the operating system, administrators can configure TLS behavior on a system-wide level, and tailor settings of individual applications to their specific needs.

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) remains safely outside the address space of the application, inaccessible to malicious parties.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.