The Secure Socket API

TLS as an Operating System Service

MARK O'NEILL, KENT SEAMONS, AND DANIEL ZAPPALA

Mark O'Neill is a PhD candidate in computer science at Brigham Young University and currently working at ManTech International. His research interests include security, networking, and artificial intelligence. His dissertation is an effort to solve modern problems in TLS by leveraging operating system and administrator control. When he's not working on research, you can find him tinkering with robots and playing StarCraft 2. Mark@markoneill.name

Kent Seamons is a Professor of Computer Science at Brigham Young University and Director of the Internet Security Research Lab. His research interests are in usable security, privacy, authentication, and trust management. His research has been funded by NSF, DHS, DARPA, and industry. He is also a co-inventor on four patents in the areas of automated trust negotiation, single sign-on, and security overlays. seamons@cs.byu.edu

Daniel Zappala is an Associate Professor of Computer Science at Brigham Young University and the Director of the Internet Research Lab. His research interests include network security and usable security. His research has been funded regularly by NSF and DHS. Daniel is an open source enthusiast and has used Linux and Emacs for about 30 years. zappala@cs.byu.edu

TLS APIs are often complex, leading to developer mistakes. In addition, even with well-written applications, security administrators lack control over how TLS is used on their machines and don't have the ability to ensure applications follow best practices. Our solution is to provide a Secure Socket API that is integrated into the well-known POSIX sockets API. This is both simple for developers to use and allows system administrators to set device policy for TLS. In this article, we both explain and demonstrate how the Secure Socket API works.

Transport Layer Security (TLS) 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 [1]. Significant efforts to catalog developer mistakes and the complexities of modern security APIs have been published in recent years. As a result, projects have emerged that reduce the size of security APIs (e.g., *libtls* in LibreSSL), enhance library security [2], and perform certificate validation checks on behalf of vulnerable applications [3, 4]. A common conclusion of these works is that TLS libraries need to be redesigned to be simpler for developers to use securely.

A related problem is that the reliance on application developers to implement security inhibits the control administrators have over their own machines. For example, administrators cannot currently dictate what version of TLS, which ciphersuites, key sizes, etc. are used by applications they install. 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.

The synthesis of these two problems is that developers lack a common, usable security API, and administrators lack control over secure connections. To address these issues, we present the Secure Socket API (SSA), a TLS API that leverages the existing standard POSIX socket API. This reduces 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, as developers merely select TLS as if it were a built-in protocol, such as TCP or UDP. Moreover, our implementation of this API enables administrators to configure TLS policies system-wide and to centrally update all applications using the API.

Secure Socket API 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`). Corresponding network operations such as `connect`, `send`, and `recv` then use the selected protocol in a manner transparent to the developer. In designing the SSA, we sought to cleanly integrate TLS into this API. 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 function signatures. Modifications to the API are acceptable only in the form of new *values* for existing parameters.
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 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.

To inform the design of the SSA, we first analyzed the OpenSSL API and its use by popular software packages. This included automated and manual assessment of 410 Ubuntu packages using TLS in client and server capacities, and assessment of the OpenSSL API itself. More details regarding our methods and results for this analysis are available at https://owntrust.org.

The API

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. Under our SSA extension, developers select TLS by specifying `IPPROTO_TLS` as the protocol in `socket`. Applications send and receive data using standard functions such as `send` and `recv`, which will be encrypted and decrypted using TLS, just as network programmers expect their data to be placed inside and removed from TCP segments under `IPPROTO_TCP`. To transparently employ TLS in this fashion, other functions of the POSIX socket API have specialized TLS behaviors under `IPPROTO_TLS` as well. In particular, `getsockopt` and `setsockopt` are used for developer configuration. A complete listing of the behaviors of the POSIX socket functions and the TLS socket options are provided in our recent paper [5].

To avoid developer misuse of TLS, the SSA is responsible for automatic management of various TLS parameters and settings, including selection of TLS versions, ciphersuites and extensions, and validation of certificates. All of these are subject to a system configuration policy with secure defaults, and customization options are exported to system administrators and developers.

To offer concrete examples of SSA use, we show code for a simple client and server below. 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 a new `AF_HOSTNAME` address family to indicate which hostname it wishes to connect to. In this case, the `connect` function performs the necessary host lookup and performs a TLS handshake with the resulting address. Alternatively, the client could have specified the hostname via a new socket option and called `connect` using traditional `INET` address families. The former method obviates the need for developers to explicitly call `gethostbyname` or `getaddrinfo`, which further simplifies their code. Either way, the SSA uses the provided hostname for certificate validation and the Server Name Indication extension to TLS. Later, the client uses `send` to transmit a plaintext HTTP request to the server, which is encrypted by the SSA before transmission. The response received is also decrypted by the SSA before placing it into the buffer provided by `recv`.

In the server case, the application binds and listens 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. Afterward, the server iteratively handles requests from incoming clients, and the SSA performs a TLS handshake with clients transparently. 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 their destinations.

```
/* 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);
return 0;
```

Listing 1: A simple HTTPS client example under the SSA. Error checks and some trivial code are removed for brevity.

```
/* Use standard IPv4 address type */
struct sockaddr_in addr;
addr.sin_family = AF_INET;
addr.sin_addr.s_addr = INADDR_ANY;
addr.sin_port = htons(443);

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

Listing 2: A simple server example under the SSA. Error checks and some trivial code are removed for brevity.

Administrator Options

Reflecting our second goal, administrator control over TLS parameters, the SSA gives administrators a protected configuration file that allows administrators to indicate their preferences for TLS versions, ciphersuites, certificate validation methodologies, extensions, and other TLS settings. These settings 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 global settings.

Our definition of administrators includes both power users as well as operating system vendors, who may wish to provide strong default policies for their users.

Developer Options

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. Linux, for example, supports 34 TCP-specific socket options to customize protocol behavior. 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 code to the implementation of `setsockopt` and `getsockopt`.

In accordance with this standard, the SSA adds a few options for `IPPROTO_TLS`. These options include setting the remote hostname, specifying a certificate chain or private key, setting a session TTL, disabling a cipher, requesting client authentication, and others. A full list is given in our recent paper [5]. Our specification of TLS options reflects a minimal set of recommendations gathered from our analysis of existing TLS use by applications, in keeping with our third design goal.

Porting Applications to the SSA

We modified the source code of four network programs to use the SSA for their TLS functionality. Two of these already used OpenSSL for their TLS functionality, and two were not built to use TLS at all. Table 1 summarizes the results of these efforts.

Both the command-line `wget` web client and the `lighttpd` web server required fewer than 20 lines of source code (Table 1), and each application was modified by a developer who had no prior experience with the code of these tools, the SSA, or OpenSSL. In addition, the modifications made it possible to remove thousands of lines of existing code. In porting these applications, most of the time spent was used to become familiar with the source code and remove OpenSSL calls.

We also modified an in-house web server and the `netcat` utility, neither of which previously supported TLS. The web server 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

Program	LOC Modified	LOC Removed	Time Taken
wget	15	1,020	5 hrs.
lighttpd	8	2,063	5 hrs.
ws-event	5	0	5 min.
netcat	5	0	10 min.

Table 1: 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.

key and certificate chain and check for corresponding errors. The TLS upgrade for `netcat` for both server and client connections required modifying five lines of code. In both cases, TLS upgrades required less than 10 minutes.

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. Any language that uses the network must interface with network system calls, either directly or indirectly. Therefore, given an implementation in the kernel, it is trivial to add SSA support to other languages.

To illustrate this benefit, we have added SSA support to three additional languages beyond C/C++: Python, PHP, and Go. Supporting these first two languages merely required making their corresponding interpreters aware of the additional constant values used in the SSA, such as `IPPROTO_TLS`. Since Go uses system calls directly and exports its own wrapper for these, we followed the same pattern by creating new wrappers for SSA functionality, which required fewer than 50 lines of code.

Implementation

We have developed a loadable Linux kernel module that implements the Secure Socket API. Source code is available at https://owntrust.org. A high-level view of a typical network application using a security library for TLS is shown in Figure 1. 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, Figure 2, 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 userspace encryption daemon. At a high-level, the kernel component is responsible for registering all `IPPROTO_TLS` functionality with the kernel and maintaining state for each TLS

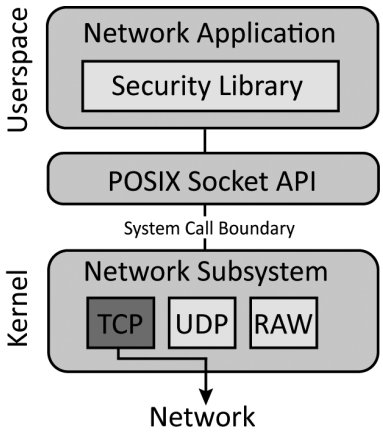

Figure 1: Data flow for traditional TLS library by network applications. The application shown is using TCP.

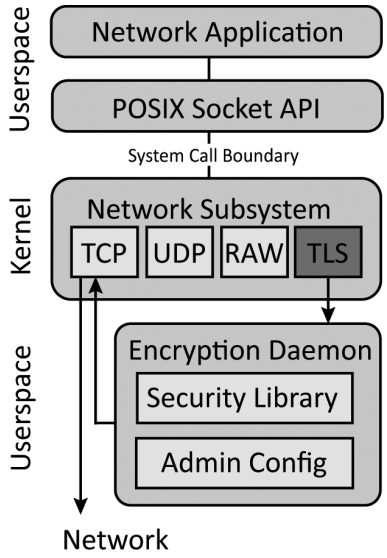

Figure 2: Data flow for SSA usage by network applications. The application shown is using the TLS, which uses TCP internally for connection-based `SOCK_STREAM` sockets.

socket. The kernel component offloads the tasks of encryption and decryption to the encryption daemon, which uses OpenSSL and obeys administrator preferences.

Note that our prototype implementation moves the use of a security library to the encryption daemon. The application interacts only with the POSIX socket API, 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, ciphersuites, and extensions to support.

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 in a pure kernel solution. This idea has been proposed within the Linux community [6] 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 back end for SSA functionality that required no userspace encryption daemon.

Discussion

Our work explores a TLS API conforming to the POSIX socket API. We reflect now on the general benefits of this approach and the specific benefits of our implementation.

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. All other socket calls 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.

Because our SSA design moves 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 [7]. For example, administrators can set preferences for TLS versions, ciphersuites, and extensions, or automatically upgrade applications to TLS 1.3 without developer patches.

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, C/C++ applications can use `IPPROTO_TLS`. 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 outside the address space of the application.

Conclusion

We feel that the POSIX socket API is a natural fit for a TLS API and hope to see it advanced through its use, new implementations, and standardization. We hope to encourage community involvement to further refine our implementation and help develop support in additional operating systems. For source code and documentation, please visit https://owntrust.org. For a more in-depth look at the SSA, see our paper presented at USENIX Security 2018 [5].

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. CNS-1528022 and the Department of Homeland Security under contract number HHSP233201600046C.

References

[1] M. Georgiev, S. Iyengar, S. Jana, R. Anubhai, D. Boneh, and V. Shmatikov, "The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '12)*, pp. 38–49: http://www.cs.utexas.edu/~shmat/shmat_ccs12.pdf.

[2] L. S. Amour and W. M. Petullo, "Improving Application Security through TLS-Library Redesign," in *Security, Privacy, and Applied Cryptography Engineering (SPACE)*, Springer, 2015, pp. 75–94.

[3] A. Bates, J. Pletcher, T. Nichols, B. Hollembaek, D. Tian, K. R. Butler, and A. Alkhelaifi, "Securing SSL Certificate Verification through Dynamic Linking," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS '14)*, pp. 394–405.

[4] M. O'Neill, S. Heidbrink, S. Ruoti, J. Whitehead, D. Bunker, L. Dickinson, T. Hendershot, J. Reynolds, K. Seamons, and D. Zappala, "TrustBase: An Architecture to Repair and Strengthen Certificate-Based Authentication," in *Proceedings of the 26th USENIX Security Symposium (USENIX Security '17)*: https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-oneill.pdf.

[5] M. O'Neill, S. Heidbrink, J. Whitehead, T. Perdue, L. Dickinson, T. Collett, N. Bonner, K. Seamons, and D. Zappala, "The Secure Socket API: TLS as an Operating System Service," in *Proceedings of the 27th USENIX Security Symposium (USENIX Security '18)*, pp. 799–816: https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-o_neill.pdf.

[6] J. Edge, "TLS in the Kernel," LWN.net: https://lwn.net/Articles/666509/; accessed: December 15, 2017.

[7] N. Mavrogiannopoulos, M. Trmac, and B. Preneel, "A Linux Kernel Cryptographic Framework: Decoupling Cryptographic Keys from Applications," in *Proceedings of the 27th ACM Symposium on Applied Computing (SAC '12)*, pp. 1435–1442: https://core.ac.uk/download/pdf/34512267.pdf.