Improving Application Security Through TLS-Library Redesign

Leo St. Amour[1] and W. Michael Petullo[2]

[1] Northeastern University
Boston, Massachusetts 02115, USA
[2] United States Military Academy
West Point, New York 10996, USA

Abstract. Research has revealed a number of pitfalls inherent in contemporary TLS libraries. Common mistakes when programming using their APIs include insufficient certificate verification and the use of weak cipher suites. These programmer errors leave applications susceptible to man-in-the-middle attacks. Furthermore, current TLS libraries encourage system designs which leave the confidentiality of secret authentication and session keys vulnerable to application flaws. This paper introduces `libtlssep` (pronounced lib·tē·el·sep), a new, open-source TLS library which provides a simpler API and improved security architecture. Applications that use `libtlssep` spawn a separate process whose role is to provide one or more TLS-protected communication channels; this child process assures proper certificate verification and isolates authentication and session keys in its separate memory space. We present a security, programmability, and performance analysis of `libtlssep`.

1 Introduction

Programs increasingly use Transport Layer Security (TLS) to protect communications. While TLS has long protected commerce and banking transactions, the protocol now routinely protects less sensitive services such as search and video streaming due to privacy concerns [23]. Researchers have even begun to investigate the notion of ubiquitous encryption [9, 14, 26]. TLS uses authentication and encryption to protect the confidentiality and integrity of communication channels, and its authentication makes use of asymmetric-key cryptography.

TLS provides server authentication through the use of X.509 identity certificates. In the most common model, some trusted Certificate Authority (CA) signs each identity certificate, ostensibly binding the public key present in the certificate to a hostname. Systems often rely on password-based client authentication which takes place after a TLS session initializes. However, TLS also supports client-side X.509-based authentication.

Permanent ID of this document: d3b9b46f59ee6212d7721c54cacb3422.
Date: July 9, 2015.

Yet attackers occasionally violate the confidentiality and integrity of communication channels despite the use of TLS. Studies by Vratonjic et al. [32], Georgiev et al. [15], and Fahl et al. [12] found that programmers consistently misuse TLS libraries in their applications. Such errors include:

(1) missing name verification,[3]
(2) trust of self-signed certificates,
(3) improper error handling,
(4) poor cipher-suite choices, and
(5) missing certificate revocation checks.

These vulnerabilities seem to arise from the Application Programming Interfaces (APIs) exported by contemporary TLS libraries. It seems that existing APIs leave too many responsibilities to the application programmer; the deceptive complexity of these steps overwhelm even those application programmers who do remember to attempt their implementation (over and over again, as they write each application). For example, Marlinspike showed how the different string encodings in X.509 and C give rise to a subtle attack on name verification [22].

Architectural choices also threaten TLS. In many cases, compromised control flow or ill information flow within an application can result in the compromise of a private cryptographic key. This is because many applications keep TLS processing and general logic in the same address space. While Heartbleed [24] attacked OpenSSL itself to compromise cryptographic keys, there are likely many more vulnerabilities present in application logic than there are present in the code included from TLS libraries. This is especially dangerous because systems often share TLS keys across several applications. For example, we counted 26 subject-alternative names plus a number of wildcards within the certificate which authenticates `bing.com` at the time of writing.

In this paper, we introduce `libtlssep` (pronounced lib·tē·el·sep), a TLS library that both simplifies the TLS API and utilizes privilege separation to increase communication security. By using `libtlssep`, an application forks a child process which is responsible for the application's TLS operations. Keeping private keys isolated in the child's separate memory space makes it more difficult for an application bug to result in a compromised key. We have released a research prototype of `libtlssep` under an open-source license, and we have made this prototype and its documentation available at `http://www.flyn.org/projects/libtlssep`.

In the following sections, we describe related work; summarize our threat model; present the design of `libtlssep`; and present security, programmability, and performance results.

2 Related work

Our survey of related work focuses on (1) pitfalls resulting from the APIs provided by existing TLS libraries, (2) efforts to improve TLS APIs, (3) existing

[3] In this paper, we refer to verifying a *name* by which we mean verifying either a certificate's subject-common name or its subject-alternative name. Such names generally represent either a host or a user.

uses of privilege separation and similar architectures, and (4) systems which provide stronger or more universal cryptographic-key isolation than `libtlssep`, albeit not without tradeoffs.

2.1 API pitfalls

Many researchers have studied the efficacy of TLS in practice [32, 15, 12]. From their work, we better understand a number of pitfalls which arise when using contemporary TLS libraries. Fahl et al. also provide evidence that the Internet is ripe with poor advice which results in programmers wrongly employing TLS libraries [13, §4.1]. We summarized the pitfalls of contemporary APIs in §1, and here we further describe this previous work.

A connection procedure which returns a TLS connection handle without first verifying the certificates involved seems to encourage omitting name verification, yet many contemporary APIs follow this pattern. Figure 1 shows in pseudo-code an example of an OpenSSL-based client. Note that a programmer could mistakenly begin calling `SSL_write` and `SSL_read` without first calling and checking the result of `SSL_get_verify_result` and `verify_name`. OpenSSL also provides a callback-type verification mechanism, but this similarly requires explicit use.

Marlinspike provided one example of why the design of X.509 makes implementing even `verify_name` difficult. Should any CA issue a certificate which contains an embedded NULL byte—such as `www.victim.com\0.attacker.com`—then it is possible that `attacker.com` could fraudulently assume the identity of `victim.com`. All that is necessary is for an application programmer to forget that NULL bytes are valid within X.509 strings but terminate C strings, such as by naïvely writing the application to use `strcmp` to compare the two as C strings. Fixed applications each duplicate strange but necessary checks such as the one found in `wget` [30]:

```
if (strlen (common_name) != (size_t) ASN1_STRING_length (sdata)) {
  /* Fail. */
}
```

Figure 2 shows in pseudo-code an example of an OpenSSL-based server. The program assumes that the client authenticates using a certificate. This requires more work from the application programmer: he would have to modify the client's logic to supply client-side certificates.

Researchers have found deployed applications which verify certificates yet trust self-signed certificates. Trusting self-signed certificates is rarely desirable except during the implementation and testing phases of development. In any case, these mistakes seem to arise from either ignorance of the dangers involved or programmers forgetting to deactivate the trust of self-signed certificates in their programs before deploying them. It is common in TLS libraries to rely on control-flow statements written by the application programmer to determine whether to honor self-signed certificates.

Additional dangers arise when the CA model itself breaks down; Durumeric et al. performed an extensive study of CA use in practice [11]. Research shows evidence that it is unreasonable for all Internet users to trust the same set of

```
1  sock = create_socket(host) // Create BSD socket.
2  method = TLSv1_2_client_method()
3  ctx = SSL_CTX_new(method)
4  SSL_CTX_set_default_verify_paths(ctx)
5  ssl = SSL_new(ctx)
6  SSL_set_fd(ssl, sock)
7  SSL_connect(ssl)
8  cert = SSL_get_peer_certificate(ssl)
9  // Programmer must explicitly check certificate:
10 if cert != NULL
11  && X509_V_OK == SSL_get_verify_result(ssl)
12  && verify_name(cert, host) { // Cert. name == host?
13    SSL_write(ssl,request,len)
14    SSL_read(ssl, response, len)
15    handle(response)
16 }
17 SSL_shutdown(ssl)
18 SSL_free(ssl)
19 close(sock)
```

Fig. 1: Pseudocode to create a TLS connection using OpenSSL. Omits error handling, except for errors related to verification. The user-defined procedures are `create_socket`, `verify_name`, and `handle`. Beurdouche et al. provide a series of similar examples [7, Listing 1–3].

CAs [31, 21]. As CAs operate in the context of many juristictions and loyalties, an individual is likely not well served by trusting all of them. Furthermore, CAs themselves have been the target of successful attacks [19]. Alternative models include PGP's web-of-trust model [34], DANE [18, 2], and certificate pinning, which closely resembles the model found in SSH [33]. Yet inspecting Figure 1 shows that programs which directly use OpenSSL each bear the responsibility of implementing verification logic from within their source code. This makes adopting emerging trust models across all applications cumbersome.

A certificate signatory ought to revoke certificates which are compromised or otherwise invalid, and part of the verification process should involve checking the revocation status of a certificate. However, existing TLS APIs permit the omission of these checks, and research has found such misuse in production applications [12, 15].

2.2 Improved APIs and static analysis

`LibreSSL` [4] aims to fix implementation errors in OpenSSL, and it also provides `libtls`. `Libtls` exports a simplified API; for example, it takes the approach of including certificate verification in the semantics of its `tls_connect` procedure, although a programmer can disable this name verification using a procedure named `tls_config_insecure_noverifyname`. Another library, `s2n` [29], has similar goals.[4]

[4] Amazon announced the `s2n` project during the final revisions of this paper.

```
1  sock = accept_connection() // Accept connection.
2  method = TLSv1_2_server_method()
3  ctx = SSL_CTX_new(method)
4  SSL_CTX_set_verify(ctx, SSL_VERIFY_PEER |
       SSL_VERIFY_FAIL_IF_NO_PEER_CERT, ignore);
5  SSL_CTX_use_certificate_file(ctx, cert, SSL_FILETYPE_PEM)
6  SSL_CTX_use_PrivateKey_file(ctx, key, SSL_FILETYPE_PEM)
7  SSL_CTX_check_private_key(ctx)
8  SSL_CTX_set_default_verify_paths(ctx)
9  ssl = SSL_new(ctx)
10 SSL_set_fd(ssl, sock)
11 SSL_accept(ssl)
12 cert = SSL_get_peer_certificate(ssl)
13 // Programmer must explicitly check certificate:
14 if cert != NULL
15  && X509_V_OK == SSL_get_verify_result(ssl)
16  && verify_user(cert) { // Cert. name permitted user?
17    SSL_read(ssl, request, len)
18    response = handle(request)
19    SSL_write(ssl,response,len)
20 }
21 SSL_shutdown(ssl)
22 SSL_free(ssl)
23 close(sock)
```

Fig. 2: Pseudocode to accept a TLS connection using OpenSSL. Omits error handling, except for errors related to verification. The user-defined procedures are `accept_connection`, `verify_user`, and `handle`.

Fahl et al. modified the Android software stack to employ a certificate-verification service which separately exists from each individual application [13, §5.2]. Moving verification to a system-wide service reduces the possibility of a programmer accidentally circumventing verification, and it also simplifies the selection and configuration of verification techniques such as certificate pinning. Programmers can also—without modifying the program—configure the verification service to enable a per-application development mode which trusts self-signed certificates. This architecture also centralizes the management of certificate warnings.

CertShim uses the `LD_PRELOAD` facility present in many Operating Systems (OSs) to assure certificate verification by replacing key TLS library procedures at runtime [3]. For example, CertShim replaces OpenSSL's `SSL_connnect` with a version which adds certificate verification to its semantics. Applications do not require modifications to take advantage of CertShim. CertShim supports a number of verification techniques, and it makes use of a single configuration point which exists separately from each application's configuration.

The NaCl library provides two common cryptographic operations: public-key authenticated encryption and signatures [6]. NaCl pursues very-high performance, side-channel-free cryptography, and the library provides a vastly sim-

pler API than contemporary cryptographic libraries. NaCl in its present form serves to replace the cryptographic-primitive procedures in TLS libraries, but it does not yet itself implement a protected network protocol. Work to build more-robust and higher-performance protocols around NaCl includes CurveCP [5] and MinimaLT [27], but these bear the cost of incompatibility with TLS.

Efforts such as the Fedora System-Wide Crypto Policy [1] seek to centralize the configuration of all cryptographic protections. This could simplify some portions of TLS configuration, although it will help less with verification because of the amount of application-specific verification code. The main beneficiary of this work will be cipher-suite selection.

SSLint uses static analysis to determine if existing programs properly use TLS-library APIs [17]. This appears complimentary to `libtlssep`, as it can help convince programmers to fix API misuse, possibly opting to migrate to a library with an improved API. The researchers behind SSLint discovered 27 previously-unknown vulnerabilities in deployed programs.

2.3 Privilege separation

Researchers have produced a number of models which increase security by using separate processes to isolate components; these designs are often described as providing *privilege separation*. The OpenSSH dæmon's privileged component can access the host's private key, open pseudo-terminals, and service change-of-identity requests [28]. Unprivileged components within OpenSSH then make indirect use of these capabilities through carefully-defined interfaces, for example by receiving pseudo-terminal file descriptors via file-descriptor passing. OpenBSD provides a framework called `imsg` [25] which aims to ease the explicit programming of communication between privileged-separated components.

The Plan 9 operating system provides a process called `factotum` which negotiates service authentication on behalf of applications [10]. `Factotum` isolates authentication keys as well as the code required for performing authentication in a separate memory space. Concentrating security code within a single program increases the programmer's ability to verify that the code is correctly written, facilitates executing the code with the least privilege required, and makes it easier to update security software. Most importantly, a logical flaw in a complicated program cannot directly lead to the compromise of an authentication key because of privilege separation. Plan 9 does not subsume from applications the work of verifying certificates.

2.4 Specialized cryptographic key isolation

Other systems provide stronger cryptographic key isolation, albeit with more intrusive requirements. One example, Mimosa [16], uses the properties of transactional memory to protect cryptographic keys from attacks originating both in user and kernel space. Yet Mimosa requires modifications to the OS kernel as well as hardware transactional memory.

Ethos is a novel OS kernel which provides digital-signature and encrypted-networking system calls [26]. This allows the kernel to universally isolate cryptographic keys from applications, and it also makes the kernel aware of the location

in memory of all cryptographic keys. Ethos is clean-slate and thus requires applications to be rewritten for all of its unique interfaces, and this burden is greater than the smaller changes required by merely porting to a new TLS API (this is a tradeoff between expediency and Ethos' stronger security properties).

Plan 9 also provides special facilities for isolating authentication keys. The system will not swap `factotum` to disk and protects `factotum`'s entry in the `/proc` filesystem. Many versions of UNIX support an encrypted swap space for similar reasons.

3 Threat model

Our threat model includes very powerful Man-in-the-Middle (MitM) attackers who can capture, modify, and deny the transmission of the messages communicated between two hosts. Specifically, our attacker can respond to the requests intended for another recipient, generate self-signed certificates, present legitimate certificates for the domains he controls, or capture legitimate certificates for the domains he does not control. Thus our goal is to use strong, *properly-applied* cryptography to provide confidentiality and integrity protections despite these attacks, namely to (1) blind the attacker to the messages we send and receive and (2) detect any attacker-manipulated traffic.

Our design removes a number of TLS misuses, and thus reduces the burden on programmers so that they can focus on the correctness of their program's core logic. It is not possible to protect against *all* programmer errors, yet we expect that the attacker will try to exploit these errors too. Such errors are orthogonal to the use of TLS, and thus they are outside of our threat model, except that we wish to avoid them compromising a cryptographic key.

We also ignore attacks on the host OS, OS access controls, the privileged account, a virtual machine monitor (if present), and hardware. We assume that the applications which make use of TLS do not do so with elevated privileges. Finally, while we are concerned about programmers selecting weak cipher suites, we ignore attacks on the TLS cryptographic protocol itself. Here there is some overlap [7], but in any case the techniques we used in `libtlssep` could likely aid in crafting libraries to support protected-networking protocols other than TLS.

4 Design of `libtlssep`

4.1 `Libtlssep` architecture

We designed our architecture to employ the isolation facilities already present in mainstream OSs to engender more robust applications. The architecture of `libtlssep` follows from the suggestions of Provos et al. [28], as it aims to aid in crafting applications which make use of privilege separation. Like SSH, `libtlssep` uses file-descriptor passing to transmit capabilities (BSD-socket connections in the case of `libtlssep`) from one process to another.

As with Plan 9's `factotum`, `libtlssep` aims to apply SSH-style privilege separation to many applications in a convenient way. `Factotum` is more general but isolates only authentication secrets; `libtlssep` spans both authentication

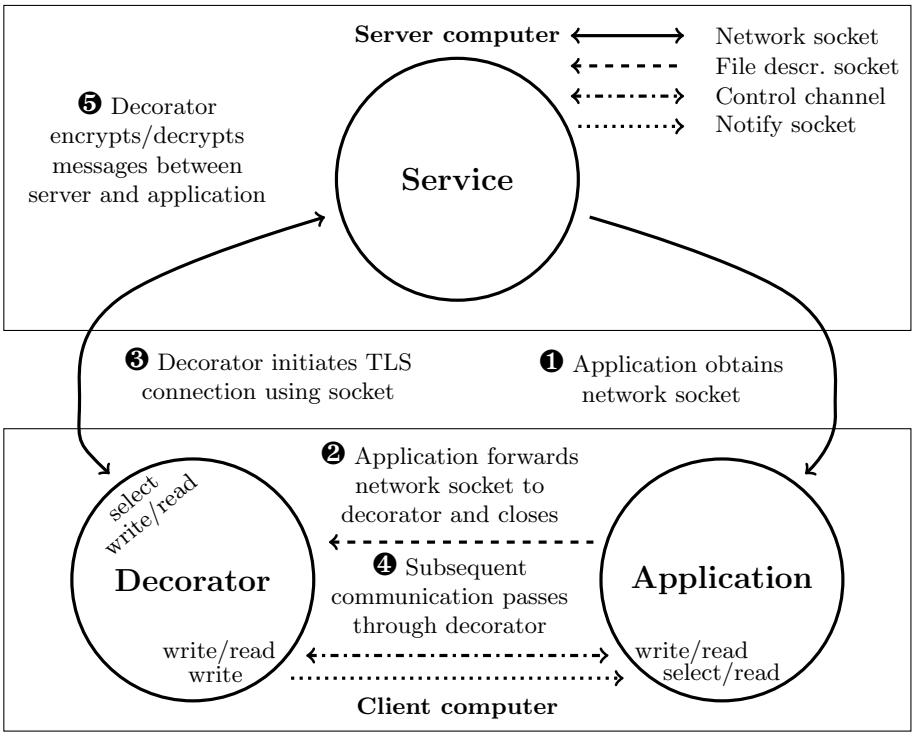

Fig. 3: Our Architecture

and encryption, isolates the session key negotiated between two parties, and provides a TLS-focused API.

`Libtlssep`'s use of a separate process also resembles Fahl's certificate-verification service, but the latter does not isolate session keys. `Libtlssep` targets C on POSIX instead of Java on Android, and while it leaves the particulars of error presentation to application programmers, an untrusted connection will result in an error code rather than proceeding.

The `libtlssep` architecture breaks applications into (at least) two processes: (1) a process containing authentication and encryption functionality, provided by `libtlssep`'s *network decorator*; and (2) a process containing program logic, provided by the application programmer. The decorator itself makes use of OpenSSL, but could be ported to any existing TLS implementation without requiring further application changes; nonetheless the decorator simplifies the use of the underlying implementation. Unlike with the direct use of OpenSSL, the decorator—like LibreSSL's `libtls`—assures the verification of certificates and hides a number of disparate OpenSSL procedure calls behind around a dozen `libtlssep` procedures. Figure 3 depicts `libtlssep`'s architecture.

`Libtlssep` uses three channels to facilitate communication between an application and its decorator: (1) a Unix-domain socket used by the application to pass file descriptors to its decorator, (2) a shared-memory- and event-file-descriptor-based control channel which allows the application to make Remote

Procedure Calls (RPCs) to its decorator, and (3) a Unix-domain notification socket which allows the application to poll for available data. The application provides yet another file descriptor—the network file descriptor—over which TLS messages flow between the decorator and remote service.

To use `libtlssep`, an application first initiates a connection with some service using the BSD-socket API. Next, the application calls the `tlssep_init` and `tlssep_connect` (or `tlssep_accept`) procedures. `Tlssep_init` executes the decorator process and initiates the control and file-decriptor-passing channels with it. `Tlssep_connect` passes the network socket and a notification socket to the decorator, and the decorator uses the network socket to initiate a TLS connection with the service. One decorator can support a number of `tlssep_connect` calls to different end points; thus two of the communication channels mentioned are per-run (i.e., the control and file-decriptor-passing channels) and two are per-TLS-connection (i.e., the network and notification sockets).

From this point on, the application communicates with the service through the decorator using `libtlssep`'s API: the application makes read and write RPCs across the control channel by calling `tlssep_read` and `tlssep_write` (possibly employing `select` on the notification socket), and the decorator wraps/unwraps the contents of these calls using TLS, passing/receiving them to/from the service.

4.2 `Libtlssep` API and configuration

API: `Libtlssep` provides around a dozen procedures which we summarize here. Most of the procedures take a `tlssep_desc_t` argument which describes an established `libtlssep` connection. The fields within the `tlssep_desc_t` structure are meant to be opaque, with the exception of the notification file descriptor which bears the field name `notificationfd`.

```
tlssep_status_t tlssep_init (tlssep_context_t *context)
```

The `tlssep_init` procedure initializes a context structure, executes the decorator process, and establishes the control and file-descriptor-passing channels described in §4.1. Upon execution, the decorator reads its configuration and begins polling the control socket.

```
tlssep_status_t tlssep_connect (tlssep_context_t *context,
                                int file_descriptor,
                                const char *expected_name,
                                char *name,
                                tlssep_desc_t *desc)
```

The `tlssep_connect` procedure provides the decorator with a network file descriptor, expected name, and the per-TLS-connection notification socket described in §4.1. After providing this information to the decorator, `libtlssep` closes the application-side copy of the network file descriptor; thereafter the application can determine if network data is available for `tlssep_read` by polling the per-TLS-connection notification socket.

Given these parameters, the decorator initiates a TLS connection and adds the given network file descriptor to the set of file descriptors it polls. Finally, the decorator verifies the certificate received from the server against `expected_name`, aborting the process if the certificate does not satisfy the configured verification engine. Upon receiving notification of a successful connection, `tlssep_connect` initializes the connection descriptor named `desc` and copies the server's true name into the buffer pointed to by `name`.

```
tlssep_status_t tlssep_accept (tlssep_context_t *context,
                               int file_descriptor,
                               const char *expected_name,
                               char *name,
                               tlssep_desc_t *desc)
```

The `tlssep_accept` procedure serves the same purpose as `tlssep_connect`, except that it implements the server side.

```
tlssep_status_t tlssep_write (tlssep_desc_t *desc,
                              const void *buf,
                              int buf_size,
                              int *bytes_written)
```

The `tlssep_write` procedure provides the decorator with a number of bytes to write on the given TLS connection.

```
tlssep_status_t tlssep_read (tlssep_desc_t *desc,
                             void *buf,
                             int buf_size,
                             int *num_read)
```

The `tlssep_read` procedure requests from the decorator a number of bytes to be read from the given TLS connection.

```
tlssep_status_t tlssep_peek (tlssep_desc_t *desc,
                             void *buf,
                             int buf_size,
                             int *num_read)
```

The `tlssep_peek` procedure serves the same purpose as `tlssep_read`, except that the returned bytes will remain in the decorator's buffer and thus remain available for subsequent reads/peeks.

```
tlssep_status_t tlssep_poll (tlssep_desc_t *desc,
                             unsigned int timeout)
```

The `tlssep_poll` procedure polls the notification socket associated with the TLS connection, blocking until the decorator has data for the application. Alternatively, a programmer can directly use UNIX's `select` system call since the `desc` structure contains the notification socket file descriptor.

```
tlssep_status_t tlssep_setnonblock (tlssep_desc_t *desc)
```

The `tlssep_setnonblock` procedure sets the mode of the decorator's network file descriptor to non-blocking.

```
tlssep_status_t tlssep_close (tlssep_desc_t *desc)
```

The `tlssep_close` procedure instructs the decorator to close the given TLS connection and remove its file descriptor from the set of file descriptors it polls. The procedure also frees any state associated with the connection.

```
tlssep_status_t tlssep_terminate (tlssep_context_t *context)
```

The `tlssep_terminate` procedure instructs the decorator to exit.

```
char *tlssep_strerror (tlssep_status_t error)
```

The `tlssep_strerror` transforms a `tlssep_status_t` status code into a human-readable string.

```
1  sock = create_socket(hostname)
2  tlssep_init(ctx)
3  status = tlssep_connect(ctx, sock, hostname, NULL, desc)
4  if TLSSEP_STATUS_OK == status {
5     tlssep_write(desc, request, len)
6     tlssep_read(desc, response, len)
7     handle(response)
8     tlssep_close(desc)
9  }
10 tlssep_terminate(desc)
```

Fig. 4: Pseudocode to create a TLS connection using `libtlssep`. Omits error handling, other than to check that the server's certificate satisfies `tlssep_connect`. The user-defined procedures are `create_socket` and `handle`.

```
1  sock = accept_connection()
2  tlssep_init(ctx)
3  status = tlssep_accept(ctx, sock, NULL, user_name, desc)
4  if TLSSEP_STATUS_OK == status && user_auth(user_name) {
5     tlssep_read(desc, request, len)
6     response = handle(request)
7     tlssep_write(desc, response, len)
8     tlssep_close(desc)
9  }
10 tlssep_terminate(desc)
```

Fig. 5: Pseudocode to accept a TLS connection using `libtlssep`. Omits error handling, other than to check that the client is authorized to connect. The user-defined procedures are `accept_connection`, `user_auth`, and `handle`.

Figure 4 shows in pseudo-code an example of a `libtlssep`-based client, and Figure 5 shows a server. In a real application, the programmer would check the status code returned from each `libtlssep` call; here we show only those checks required to perform authentication. §5.1 will describe the security advantages of `libtlssep`'s API.

Configuration: CertShim provided the inspiration for `libtlssep`'s configuration engine. Figure 6 lists a sample `libtlssep` configuration as is typically found at `/etc/tlssep-decorator-`*`api-version`*`.cfg`, where *`api-version`* represents the major and minor version numbers of `libtlssep`. Lines 1–3 specify the global configuration parameters, in this case the path to a certificate and private key as well as the default certificate-trust model.

The application-specific statement beginning on line 5 overrides the configuration when `tlssep-decorator` acts on behalf of `/usr/bin/my-prototype` so that the program chains two verification techniques: the traditional CA model and self-signed certificates, with the latter presumably supported for development purposes. Here the meaning of the `enough` parameter resembles CertShim's `vote`: satisfying *one* of either CA or self-signed verification is sufficient for this application.

Had the administrator set `enough` to `2`, the application would require that *both* verifications be successful; in the absence of an `enough` parameter, `tlssep-decorator` will enforce *all* of the specified verification techniques. An administrator could select other trust models here without making any changes to application source code.

```
1  certpath = "/etc/pki/tls/certs/cert.pem";
2  privkeypath = "/etc/pki/tls/certs/key.pem";
3  verification = ( "ca" );

5  programs = ({
6     path = "/usr/bin/my-prototype";
7     verification = ( "ca", "self-signed" );
8     enough = 1;
9  })
```

Fig. 6: Sample `libtlssep` configuration.

5 Security, programmability, and performance

5.1 Security benefits of `libtlssep`'s API and architecture

Table 1 summarizes the security advantages of `libtlssep` which we further describe here. `Libtlssep` contributes to application robustness for two reasons: (1) it has a simple API which we designed to provide clear failure semantics, and (2) it results in applications which make use of privilege separation to protect secret cryptographic keys.

Our design represents a tradeoff: for example, combining another TLS library with OpenBSD's `imsg` would provide more flexibility, but such a composi-

Table 1: Comparison of OpenSSL and `libtlssep`.

	OpenSSL	`libtlssep`
Certificate verification	Left to application programmer; trust model (including trust of self-signed certificates) embedded in application logic	Follows from semantics of library; trust model selected by configuration
Name verification	Application programmer must check that the certificate's name matches the expected name	Follows from semantics of library
Error reporting	Inconsistent API [15, §4.1]	Consistent API
Key isolation	Key compromise follows from application compromise	Architecture isolates keys in separate memory space
Configuration	Each application has its own configuration mechanism	Single configuration point for all applications
OS access controls	OS has difficulty discerning between encrypted and cleartext connections	OS can restrict applications such that they can only perform network reads and writes through decorator
Cipher suite choices	Left to application programmer; includes `null` cipher	Library designers choose cipher suite

tion requires the programmer to design a privilege-separation architecture. With `libtlssep`, programmers benefit from the architecture we designed to protect cryptographic keys without needing to reason about privilege separation.

`Libtlssep`'s API promotes better application security. Recall Figures 1, 2, 4, and 5 which show examples of using OpenSSL and `libtlssep`. Figure 1 shows that a client application programmer who makes direct use of OpenSSL must call a number of procedures to set up the TLS connection. Most significantly, explicit code is required to verify the peer certificate involved in the connection; this involves obtaining the peer certificate using `SSL_get_peer_certificate`, verifying it through a call to `SSL_get_verify_result`, and further checking the certificate's name by implementing and calling `verify_name`. We discussed in §2 Marlinspike's attack on subtle flaws in `verify_name`-like procedures, and contemporary TLS APIs cause such procedures to be repeated across many applications.

Figure 4 shows that `libtlssep` requires fewer procedure calls, and thus allows less ill composition. Here the programmer does not have to explic-

itly call a verification routine. Instead, verification follows from the semantics of `tlssep_connect` (or `tlssep_accept`), as with LibreSSL's `tls_connect`. `Libtlssep` does not return a valid TLS connection handle if verification fails. By using `libtlssep` instead of directly using OpenSSL, an application remains simpler, because `libtlssep` absolves the application programmer of the responsibility of verification. `Libtlssep` also makes error handling more clear as its procedures report errors in a consistent manner, unlike many existing APIs [15, §4.1].

With `libtlssep`, all network messages subsequent to the initial connection establishment pass through the decorator. The decorator isolates both long-term authentication keys and session keys in its own address space. This reduces the likelihood that an application compromise will result in the compromise of a cryptographic secret. This design is intended to address issues which stem from a combination of (1) implementation flaws which allow for applications to be compromised and (2) design flaws which allow long-term keys to exist in an application's memory space. We do not claim to fix all attacks, but our implementation will help with those that exploit application code to retrieve long-term keys.

In other architectures, both verification code and configuration settings are duplicated throughout a number of applications. Bugs fixed in one application are left latent in others, and administrators must learn each application's TLS-configuration syntax. With `libtlssep`, the API and decorator consolidates verification code, ensures applications cannot ignore verification failures, and consolidates trust-model configuration. Programmers are accustomed to deploying different configuration files than those used while developing their software, so this will reduce the likelihood of deploying an application which trusts self-signed certificates. Furthermore, upgrading `libtlssep` and modifying the library's configuration file can add new certificate trust models without modifying applications. This can also centralize efforts to address emerging threats—such as with the triple-handshake attack [8]—which with other libraries require updates to each application.

`Libtlssep`'s decorator will exist in a filesystem with its setuid bit set. This ensures that the decorator runs as a different user than the application. The decorator's user should have special read access to the appropriate cryptographic keys, but should not necessarily have full superuser privileges.

`Libtlssep`'s architecture allows OSs to better constrain applications which make use of the library. For example, Security-Enhanced Linux (SELinux) or another fine-grained access-control system could forbid an application from reading or writing cleartext network connections, instead permitting only TLS-protected communication through the `libtlssep` decorator. This forces applications to communicate over the network only in an encrypted manner. Current architectures make it difficult to discern between encrypted and cleartext connections from within OS access controls. Existing techniques rely on weaker transport-layer-port filtering or attempts at runtime packet inspection.

Our design does not allow programmers to pick the cipher suites their applications use. This allows `libtlssep` to avoid cryptographic disasters such as weak ciphers, disabled cryptography, or ill-composed cryptographic primitives [6].

5.2 Programmability

To assess the programmability of `libtlssep`, we ported two common applications: the `wget` client [30] and the `lighttpd` [20] server.

Porting `wget` required the addition of 231 lines of code—31 of which were comments—and the removal of three lines (`wget` totals around 39,000 lines). A number of these additions involved properly implementing error handling and following good programming practices. We benefited from the fact that `wget` already supports multiple TLS backends, so our additions took the form of a `libtlssep` backend and modified only two source files. The `libtlssep` backend comprises of 159 lines of code while the OpenSSL backend consumes 590.

Porting `lighttpd` required the addition of 352 lines and the removal of 15 lines (`lighttpd` totals around 40,000 lines). `Lighttpd` was not written to support multiple TLS backends, which slightly added to the difficulty of our port. Here we ended up replacing OpenSSL procedure calls with `libtlssep` procedure calls in seven source files.

Modifying `wget` and `lighttpd` to use our library shows that existing applications—both client- and server-side—can easily gain the security benefits provided by `libtlssep`. In both cases, we completed the port without having previously studied the application's source code. The use of CertShim with existing applications requires even less effort, but CertShim does not provide the architectural security benefits of `libtlssep`. New applications will immediately benefit from choosing `libtlssep`'s simpler API.

5.3 Performance

To evaluate `libtlssep`'s performance, we measured latency and throughput while comparing `libtlssep` with pure OpenSSL. We made use of a computer with a 3.4-GHz four-core Intel Core i7-3770 processor and 32 GB of memory. We ran our tests by requesting data from a local HTTPS server using the loopback interface; thus our results amplify the performance differences between `libtlssep` and OpenSSL because they omit real network latency.

For testing purposes, we created four HTTPS clients: for each of OpenSSL and `libtlssep`, we implemented a latency- and throughput-testing client. Lighttpd 1.4.36 (compiled to use pure OpenSSL, not `libtlssep`) provided the HTTPS server for our test clients. Each benchmark uses the same cipher suite: ephemeral elliptic-curve Diffie-Hellman, RSA, 128-bit AES, and SHA-256. We also performed tests using `wget` and `lighttpd`, each compiled to use both pure OpenSSL and `libtlssep`.

Latency performance: Each of our latency benchmarks repeats the process of initiating a TLS connection, reading one byte, and then closing the connection. We measured the time that it took each application to complete 10,000 iterations. Table 2a summarizes the results of this experiment. We present the results of 10 full runs, along with the mean and standard deviation. The OpenSSL implemen-

Table 2: Latency and throughput measurements. Both client and server ran on the same machine and communicated using the loopback interface.

#	Runtime (seconds) OpenSSL	`libtlssep`
1	49.341	50.903
2	49.474	51.178
3	49.112	50.783
4	49.358	50.945
5	49.457	50.604
6	50.563	51.212
7	49.818	50.594
8	49.764	51.075
9	49.563	51.028
10	49.353	50.616
μ	49.580	50.894
σ	0.403	0.235

(a) Runtime of 10,000 serial connections using OpenSSL and `libtlssep`.

#	Runtime (seconds) OpenSSL			`libtlssep`		
	10^1 MB	10^2 MB	10^3 MB	10^1 MB	10^2 MB	10^3 MB
1	0.061	0.193	1.517	0.071	0.205	1.625
2	0.061	0.193	1.517	0.070	0.205	1.557
3	0.061	0.193	2.420	0.070	0.206	1.589
4	0.061	0.193	1.521	0.079	0.294	1.585
5	0.061	0.194	1.517	0.070	0.212	1.841
6	0.061	0.194	1.521	0.070	0.212	1.557
7	0.061	0.194	1.559	0.070	0.294	2.446
8	0.061	0.193	1.519	0.071	0.206	1.554
9	0.061	0.194	1.518	0.070	0.207	1.563
10	0.061	0.194	2.417	0.070	0.295	2.456
μ	0.061	0.194	1.703	0.071	0.234	1.777
σ	0.000	0.000	0.378	0.003	0.042	0.365

(b) Single download time of file sizes indicated using OpenSSL and `libtlssep`.

tation had an average runtime of 49.580 seconds with a standard deviation of 0.403. The `libtlssep` implementation had an average runtime of 50.894 seconds with a standard deviation of 0.235 seconds. On average, `libtlssep` initiates TLS connections at 97.4% the rate measured with pure OpenSSL.

Throughput performance: Our throughput benchmarks read files of varying sizes over a TLS connection. Each creates a single connection, reads 1,024 MB at a time until the entire file is read, and then closes the connection. We measured the time that it took each application to download 10 MB, 100 MB, and 1,000 MB files.

Table 2b summarizes the results of this experiment. For the 1,000 MB file, the pure OpenSSL implementation took an average of 1.703 seconds with a standard deviation of 0.378, while the `libtlssep` implementation took an average of 1.777 seconds with a standard deviation of 0.365.

Based on these results, the throughput of `libtlssep` is 95.8% of that measured with pure OpenSSL. The slight difference is due to the added overhead of scheduling an additional process as well as the additional `memcpy`s and RPC-related shared-memory communication involved. `Libtlssep`'s throughput during our experiments exceeded 4,610 Mb/s.

We also performed benchmarks using both our `libtlssep` and the upstream-OpenSSL versions of `lighttpd` and `wget`. Here we used variations of the follow-

Table 3: Single download time of a 1,000 MB file using the OpenSSL and `libtlssep` versions of `wget` and `lighttpd`.

	Runtime (seconds)			
#	**OpenSSL server OpenSSL client**	**OpenSSL server `libtlssep` client**	**`libtlssep` server OpenSSL client**	**`libtlssep` server `libtlssep` client**
1	1.601	2.643	2.066	2.722
2	1.597	2.565	2.055	2.704
3	1.599	2.566	2.247	2.732
4	1.598	2.559	2.040	2.645
5	1.598	2.584	2.056	2.865
6	1.600	2.590	2.052	2.581
7	1.597	2.565	2.103	2.643
8	1.596	2.563	2.051	2.977
9	1.608	2.564	2.054	2.736
10	1.829	2.619	2.059	2.855
μ	1.622	2.582	2.078	2.746
σ	0.073	0.028	0.062	0.120

ing command (note that the `libtlssep` version of `wget` presently ignores the `--no-check-certificate` option):

```
time wget --quiet --no-http-keep-alive --no-check-certificate \
  -O /dev/null https://127.0.0.1/1000M
```

We summarize our `lighttpd`-to-`wget` results in Tables 3 and 4. The former table contains measurements of a single 1,000 MB download, and the latter table contains measurements of three simultaneous 1,000 MB downloads. A single serial `libtlssep`-to-`libtlssep` download provides approximately 59% the throughput of its pure-OpenSSL counterpart when transfering over our computer's loopback interface. This rate would benefit from increasing the size of the buffers used within `lighttpd` and `wget` to reduce the number of RPCs `libtlssep` must invoke to transfer data (also recall that our previous experiments used `libtlssep` only on one side of the connection). Simultaneous transfers fare better; here `libtlssep` approaches within 68% of OpenSSL's throughput. This performance would also benefit from tuning the buffer sizes within `lighttpd` and `wget`.

6 Conclusion

`Libtlssep` provides application programmers with a simpler API and more secure design for adding TLS support to their applications. `Libtlssep` is less ambitious than other projects; it exists between contemporary TLS libraries and projects such as NaCL [6] and MinimaLT [27]. `Libtlssep` serves as an easy-to-integrate, near-term replacement for existing TLS libraries. Nonetheless, `libtlssep` provides better isolation of cryptographic secrets and reduces the number of pitfalls faced by network programmers.

Table 4: Total download time of three simultaneous transfers of a 1,000 MB file from OpenSSL/`libtlssep` `lighttpd` to OpenSSL `wget`s.

	Runtime (seconds)	
#	**OpenSSL server**	**`libtlssep` server**
1	4.466	6.449
2	4.470	6.203
3	4.479	6.558
4	4.545	6.621
5	4.622	6.809
6	4.438	6.667
7	4.428	6.725
8	4.432	6.445
9	4.519	6.494
10	4.455	6.310
μ	4.485	6.528
σ	0.061	0.187

Future work on `libtlssep` will include further performance optimizations, a review of the library's source code, and additional application ports. Previous performance improvements came from replacing our use of Open Network Computing (ONC) RPC with a custom RPC implementation, moving from a UNIX-socket-based to a shared-memory-based RPC channel, and reusing a single decorator process across multiple connections within an application. `Libtlssep`'s decorator would also benefit from an implementation in a strongly typed language such as Go. Once we are satisfied with our implementation and API we will announce a stable release; our research prototype is already available at `http://www.flyn.org/projects/libtlssep`.

Acknowledgments

We thank Suzanne Matthews, Kyle Moses, and Christa Chewar for their comments on our early work, our anonymous referees for comments on subsequent drafts, and the United States Military Academy for their support. We are also grateful to Colm MacCárthaigh who encouraged us to pursue using shared memory to improve the performance of `libtlssep`'s RPC channel.

References

[1] Fedora system-wide crypto policy. `http://fedoraproject.org/wiki/Changes/CryptoPolicy` [Accessed Mar 22, 2014]

[2] Barnes, R.L.: DANE: Taking TLS authentication to the next level using DNSSEC. IETF Journal (Oct 2011), `http://www.internetsociety.org/articles/dane-taking-tls-authentication-next-level-using-dnssec` [Accessed Jun 22, 2015]

[3] Bates, A., Pletcher, J., Nichols, T., Hollembaek, B., Tian, D., Butler, K.R., Alkhelaifi, A.: Securing SSL certificate verification through dynamic linking. In: Pro-

ceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. pp. 394–405. CCS '14, ACM, New York, NY, USA (2014)
[4] Beck, B.: LibreSSL: The first 30 days and the future (May 2014), presentation at the 11th BSDCan Conference
[5] Bernstein, D.J.: CurveCP: Usable security for the Internet. CurveCP: Usable security for the Internet, http://curvecp.org [Accessed Jul 9, 2015]
[6] Bernstein, D.J., Lange, T., Schwabe, P.: The security impact of a new cryptographic library. In: International Conference on Cryptology and Information Security in Latin America. Lecture Notes in Computer Science, vol. 7533, pp. 159–176. Springer (2012)
[7] Beurdouche, B., Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Kohlweiss, M., Pironti, A., Strub, P.Y., Zinzindohoue, J.K.: A messy state of the union: Taming the composite state machines of TLS. In: Proc. IEEE Symp. Security and Privacy. IEEE Computer Society Press, Washington, DC, USA (May 2015)
[8] Bhargavan, K., Lavaud, A., Fournet, C., Pironti, A., Strub, P.: Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS. In: Proc. IEEE Symp. Security and Privacy. pp. 98–113. IEEE Computer Society Press, Washington, DC, USA (May 2014)
[9] Bittau, A., Hamburg, M., Handley, M., Mazières, D., Boneh, D.: The case for ubiquitous transport-level encryption. In: Proceedings of the 19th USENIX Security Symposium. USENIX Association, Berkeley, CA, USA (Aug 2010)
[10] Cox, R., Grosse, E., Pike, R., Presotto, D., Quinlan, S.: Security in Plan 9. In: Proc. of the USENIX Security Symposium. pp. 3–16. USENIX Association, Berkeley, CA, USA (2002)
[11] Durumeric, Z., Kasten, J., Bailey, M., Halderman, J.A.: Analysis of the HTTPS certificate ecosystem. In: Proceedings of the 2013 Conference on Internet Measurement. pp. 291–304. IMC '13, ACM, New York, NY, USA (2013)
[12] Fahl, S., Harbach, M., Muders, T., Smith, M., Baumgärtner, L., Freisleben, B.: Why Eve and Mallory love Android: an analysis of Android SSL (in)security. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 50–61. ACM, New York, NY, USA (2012)
[13] Fahl, S., Harbach, M., Perl, H., Koetter, M., Smith, M.: Rethinking SSL development in an appified world. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. pp. 49–60. CCS '13, ACM, New York, NY, USA (2013)
[14] Electronic Frontier Foundation: HTTPS everywhere, https://www.eff.org/https-everywhere [Accessed Aug 26, 2013]
[15] Georgiev, M., Iyengar, S., Jana, S., Anubhai, R., Boneh, D., Shmatikov, V.: The most dangerous code in the world: validating SSL certificates in non-browser software. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp. 38–49. CCS '12, ACM, New York, NY, USA (2012)
[16] Guan, L., Lin, J., Luo, B., Jing, J., Wang, J.: Protecting private keys against memory disclosure attacks using hardware transactional memory. In: Proc. IEEE Symp. Security and Privacy. IEEE Computer Society Press, Washington, DC, USA (May 2015)
[17] He, B., Rastogi, V., Cao, Y., Chen, Y., Venkatakrishnan, V., Yang, R., Zhang, Z.: Vetting SSL usage in applications with SSLint. In: Proc. IEEE Symp. Security and Privacy. IEEE Computer Society Press, Washington, DC, USA (May 2015)
[18] Hoffman, P., Schlyter, J.: RFC 6698: The DNS-based Authentication of Named Entities (DANE) Transport Layer Security (TLS) protocol: TLSA. http://www.

ietf.org/rfc/rfc6698.txt [Accessed Jun 22, 2015] (Aug 2012), status: PROPOSED STANDARD
[19] IOerror: DigiNotar damage disclosure. The Tor Blog (Sep 2011), https://blog.torproject.org/blog/diginotar-damage-disclosure [Accessed May 20, 2015]
[20] Kneschke, J., et al.: lighttpd, http://www.lighttpd.net/ [Accessed Jun 22, 2015]
[21] Leavitt, N.: Internet security under attack: The undermining of digital certificates. Computer 44(12), 17–20 (Dec 2011)
[22] Marlinspike, M.: Null-prefix attacks against SSL/TLS certificates. Presentation at Black Hat USA (Jul 2009), http://www.blackhat.com/presentations/bh-usa-09/MARLINSPIKE/BHUSA09-Marlinspike-DefeatSSL-PAPER1.pdf [Accessed Jun 22, 2015]
[23] Naylor, D., Finamore, A., Leontiadis, I., Grunenberger, Y., Mellia, M., Munafò, M., Papagiannaki, K., Steenkiste, P.: The cost of the 'S' in HTTPS. In: Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies. pp. 133–140. CoNEXT '14, ACM, New York, NY, USA (2014)
[24] NIST National Vulnerability Database: CVE-2014-0160. http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160 (Dec 2013), [Accessed Apr 15, 2014]
[25] OpenBSD manual pages: imsg_init(3), http://www.openbsd.org/cgi-bin/man.cgi/OpenBSD-current/man3/imsg_init.3 [Accessed Jul 8, 2015]
[26] Petullo, W.M., Solworth, J.A.: Simple-to-use, secure-by-design networking in Ethos. In: Proceedings of the Sixth European Workshop on System Security. EUROSEC '13, ACM, New York, NY, USA (Apr 2013)
[27] Petullo, W.M., Zhang, X., Solworth, J.A., Bernstein, D.J., Lange, T.: MinimaLT: Minimal-latency networking through better security. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security. CCS '13, ACM, New York, NY, USA (Nov 2013)
[28] Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: Proc. of the USENIX Security Symposium. pp. 231–242. USENIX Association, Berkeley, CA, USA (Aug 2003)
[29] Schmidt, S.: Introducing s2n, a new open source TLS implementation. Amazon Web Services Security Blog (Jun 2015), https://blogs.aws.amazon.com/security/post/TxCKZM94ST1S6Y/Introducing-s2n-a-New-Open-Source-TLS-Implementation [Accessed Jul 1, 2015]
[30] Scrivano, G., et al.: wget, http://www.gnu.org/software/wget/ [Accessed Jun 22, 2015]
[31] Soghoian, C., Stamm, S.: Certified lies: detecting and defeating government interception attacks against SSL. In: Proceedings of the 15th international conference on Financial Cryptography and Data Security. pp. 250–259. FC'11, Springer-Verlag, Berlin, Heidelberg (2012)
[32] Vratonjic, N., Freudiger, J., Bindschaedler, V., Hubaux, J.P.: The inconvenient truth about web certificates. In: Proceedings of the 10th Workshop on the Economics of Information Security (Jun 2011)
[33] Ylonen, T.: SSH—secure login connections over the Internet. In: Proc. of the USENIX Security Symposium. pp. 37–42. USENIX Association, San Jose, California (1996)
[34] Zimmermann, P.R.: The Official PGP Users Guide. MIT Press, Boston, Massachusetts, U.S.A. (1995)