Foundations of Cybersecurity Project Report

Lorenzo Catoni Leonardo Giovannoni Marco Lucio Mangiacapre

Contents

1 Introduction

We decided to split our project in two different logic layers (somewhat similar to what https does): one secure socket layer that handles all the cryptographic functionalities and an application layer that handles the business logic.

The Simple Secure Socket Layer (S3L from now on) is a protocol implemented in the SecureDataChannel class, which acts like a proxy on a channel, in particular in our case for the classical tcp socket. It manages all the cryptographic functionalities in a transparent way with regard to the user, the interface that it provides is very similar to the one of a classical socket.

The Application Layer Protocol (ALP from now on) handles the various command as per specifications, without explicit knowledge of the underlying socket implementation, it can work on classical unencrypted channels as well as on S3L channels (in our case, SocketChannel instances).

The details about the cryptographic protocol and the application protocol are described respectively in chapter 2 and 3. In chapter 4 some implementation details are discussed. Finally in chapter 4 we examine some final remarks.

2 The cryptographic protocol

2.1 Overview

Figure 1 shows the sequence diagram for our cryptographic protocol: it is heavily inspired from the **Station-to-Station** protocol. During the handshake the identity of both the client and the server is verified and two session keys are negotiated: one for encryption and one for authentication.

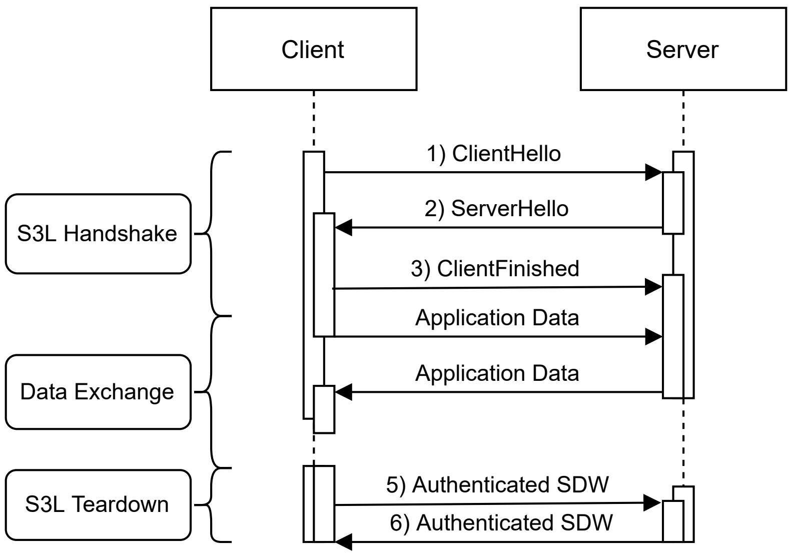


Figure 1: Sequence diagram for the cryptographic protocol

In the **Client Hello** packet the client sends the following information:

- Ephemeral Diffie-Hellman client public key

The server sends a **Server Hello** packet with the following information:

- Ephemeral Diffie-Hellman server public key
- The server's certificate

- Signature of the two Diffie-Hellman public keys

The server can already calculate the shared key with Diffie-Hellman, the shared key will be hashed with sha256 and the first 16 bytes will be used to encrypt messages, the last 16 to authenticate them. Now the client has to verify that the server's certificate is valid and that the signature is valid, after that it can derive the shared key in the same way as the server. Then it sends a **Client Finished** packet with:

- The user id
- The iv that will be used for the symmetric cipher later
- Signature of the two Diffie-Hellman public keys
- An hmac with the shared key of the above fields

From now on all messages will be encrypted with AES-CTR-128 and authenticated with HMAC-SHA-256. The generic structure for the protocol packets can be seen in figure 2.

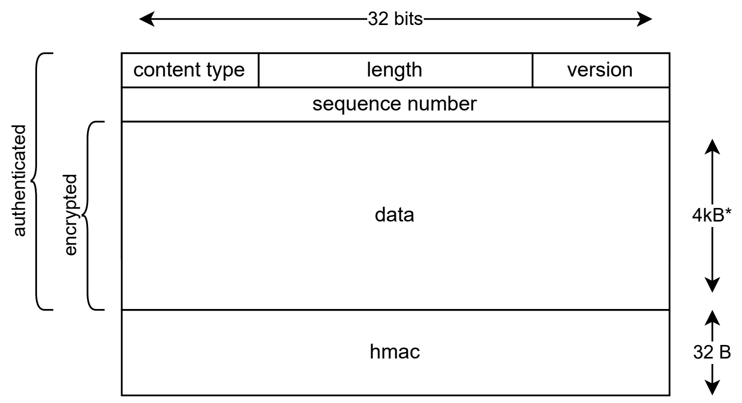


Figure 2: Layout of packets exchanged in S3L

In figure 3 the handshake steps can be seen in detail.

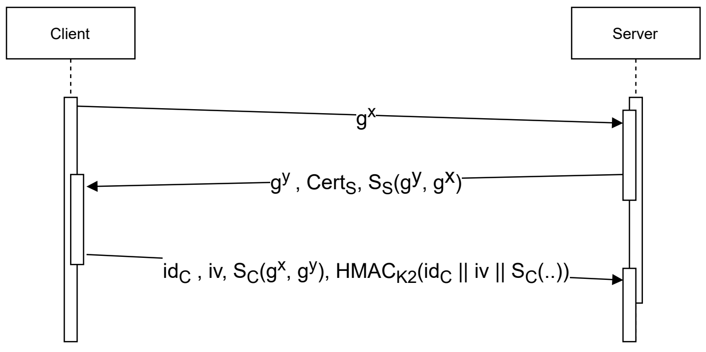

Client
Server
g^x
g^y , $Cert_S$, $S_S(g^y, g^x)$
id_C , iv, $S_C(g^x, g^y)$, $HMAC_{K2}(id_C \| iv \| S_C(..))$

Figure 3: Detailed view of the handshake steps

2.2 Meeting the security requirements

The security requirements where **Perfect Forward Secrecy**, **encryption and authentication** of the entire session and resistance to **reply attacks**.

Perfect Forward Security is guaranteed by the Ephemeral Diffie-Hellman key exchange, the exchange is authenticated to prevent the possibility of MITM attacks.

Resistance to **reply attacks** in the handshake is given by the use of Ephemeral Diffie-Hellman, during data exchange such attacks are not possible because every packet has an incremental counter called `sequence number` that is authenticated together with the rest of the S3L packet.

Encryption and Authentication is achieved with two different shared keys, used for AES-CTR-128 and HMAC-SHA-256 respectively.

3 The application layer protocol

The protocol used to exchange data between client and server at the application level appears in the code as ALP. It is based on a header, in xml format, with a fixed size of 1024 bytes, containing metadata relating to the body which constitutes the actual content of the message as in http. In particular, the fields contained in the header are the following:

- **reply**: this field is used in the reply and indicates the result of the request. A subset of the http conventions are used
- **content_type**: this field has the function of specifying the content of the body, in this version there are two options:
  - **text/xml**: the body contains a string representing an object in xml format
  - **binary/file_chunk**: the body contains bytes representing the chunk of a file
- **endpoint**: represents the endpoint that the client wants to call, this will correspond to a method with the same name as the controller
- **method**: field not used, as all requests are of type "POST", it could be used for optimizations.
- **body_size**: represents the size of the body in bytes
- **range_begin**: optional field that indicates the offset of the file chunk that is the subject of the operation
- **file_path**: optional field that specifies the path for the file chunk that is the subject of the operation

In code excerpt 1 we can see a ALP header for a request to the `ListFiles` endpoint.

```
<ALPHeader>
       <reply></reply>
       <content_type>text/xml</content_type>
       <endpoint>ListFiles</endpoint>
       <method></method>
       <body_size>91</body_size>
       <range_begin class_id="1" tracking_level="0" version="1">
              <initialized>0</initialized>
       </range_begin>
       <file_path class_id="2" tracking_level="0" version="1">
              <initialized>0</initialized>
       </file_path>
</ALPHeader>
```

Listing 1: ALP header for a request to the `ListFiles` endpoint

Leaving aside for the moment the management of file transmission, the protocol requires the client to build an object compatible with the controller method declared as an endpoint, then it serializes the object and it sends it as a string (byte vector without terminator). The server will then deserialize the string interpreting it as xml and it will pass the deserialized object to the c method specified in the header. This handler will execute the routine requested by the client and will eventually return a value.

In the event that a file has to be sent, the client will have to read the file and send it in chunks. The body of the message is a byte vector containing only the chunk of the file in binary format, the server will then write the chunk of the file to the path and offset specified in the header.

4 C++ implementation details

We used C++20 with gcc 11.2 as compiler, the coding style is inspired by Google C++ style. In addition to OpenSSL, we also used the boost library that allowed us to use many features that the C++ standard library doesn't have (yet). This is a well known library used in many important projects (e.g. MongoDB), and with no doubt the most important library excluding the standard library. Boost library is huge and in particular we used boost asio, which is a cross-platform asynchronous library for networking (for simplicity, we used that in synchronous mode), which as of 2022 is experimental for the standard library; we also used the boost serialization library to easily serialize C++ structures and to parse XML configuration files.

We handled with care the following issues related to secure coding, canonicalization and injection prevention. Our protocol implementation isn't tied to network or to OpenSSL structures, instead we use the Channel interface to manage the typical operations of, for instance, a socket. A Channel is an interface where you can read, write, check whether is closed and close. This allowed us to test the protocol and the code first in a LocalChannel and then in the SocketChannel.

4.1 Memory management

Since we developed our project in C++ but the openssl api is written in C we had to take extra measures in order to avoid memory leaks in case of error, code except 2 is an example to illustrate this issue (there are no checks for the return values of the openssl functions for the sake of simplicity). The problem is that, if the instruction at line 9 throws a runtime exception, the free at line 11 will never be executed and we just leaked some memory, this happens because we are mixing C and C++ code. The most simple way to handle this issue is to expoit RAII idiom, allowing the compiler to place cleanup routine when needed.

One correct solution is illustrated in code excerpt 3. In this case we wrap the `bio` object in a `std::unique_ptr`, now the destructor (`BIO_free` in this case) will be called when the object goes out of scope, handling correctly the case of exceptions.

```
1  std::vector<u8> GetPubkeyAsBytes(EVP_PKEY *pubkey) {
2    BIO *bio = BIO_new(BIO_s_mem());
3
4    PEM_write_bio_PUBKEY(bio, pubkey);
5
6    u8 *ptr;
7    long size = BIO_get_mem_data(bio, &ptr);
8
9    auto rv = std::vector<u8>(ptr, ptr + size);
10
11   BIO_free(bio);
12
13   return rv;
14 }
```

Listing 2: Wrong management of heap memory

```
std::vector<u8> GetPubkeyAsBytes(EVP_PKEY *pubkey) {
  BIO *bio = BIO_new(BIO_s_mem());
  OPENSSLCHECKALLOC(bio);
  auto biobox = std::unique_ptr<BIO, decltype(&BIO_free)>(bio, BIO_free);

  OPENSSLCHECK(PEM_write_bio_PUBKEY(biobox.get(), pubkey));

  u8 *ptr;
  long size = BIO_get_mem_data(biobox.get(), &ptr);

  auto rv = Vec<u8>(ptr, ptr + size);

  return rv;
}
```

Listing 3: Correct management of heap memory

4.2 Secure disposal of session secrets

In order to correctly erase from memory session secrets when they are not needed anymore we created a class `SecVec<T>` which behaves like a `std::vector<T>` but upon destruction the content is set to 0 in a way that is safe with respect to compiler optimizations. Code excerpt 4 shows the implementation of `SecVec<T>` using `std::vector<T>` plus a custom `allocator`. This code is the same used by Botan, a cryptographic library written in modern C++.

```
/* memset to 0 in a compiler optimization resistant way */
inline void secure_scrub_memory_func(void *ptr, size_t n) {
  volatile auto *p = reinterpret_cast<volatile uint8_t *>(ptr);
  for (size_t i = 0; i != n; ++i)
    p[i] = 0;
}

template<typename T>
class secure_allocator {
 public:
  /**
   * Assert exists to prevent someone from doing something that will
   * probably crash anyway (like secure_vector<non_POD_t> where ~non_POD_t
   * deletes a member pointer which was zeroed before it ran).
   */
  static_assert(std::is_integral<T>::value, "secure_allocator supports only integer types");
  typedef T value_type;
  typedef std::size_t size_type;

  secure_allocator() noexcept = default;

  secure_allocator(const secure_allocator &) noexcept = default;

  secure_allocator &operator=(const secure_allocator &) noexcept = default;

  ~secure_allocator() noexcept = default;

  template<typename U>
  secure_allocator(const secure_allocator<U> &) noexcept {}

  T *allocate(std::size_t n) {
    return static_cast<T *>(operator new(n * sizeof(T)));
  }

  void deallocate(T *p, std::size_t n) {
    if (p == nullptr)
      return;

    secure_scrub_memory(p, n * sizeof(T));
    ::operator delete(p);
  }
};

template<typename T, typename U>
inline bool
operator==(const secure_allocator<T> &, const secure_allocator<U> &) { return true; }

```

```cpp
template<typename T, typename U>
inline bool
operator!=(const secure_allocator<T> &, const secure_allocator<U> &) { return false; }

/* SecVec container */
template<typename T>
using SecVec = std::vector<T, secure_allocator<T>>;
```

Listing 4: Implementation of `SecVec<T>`

4.3 User input sanitization

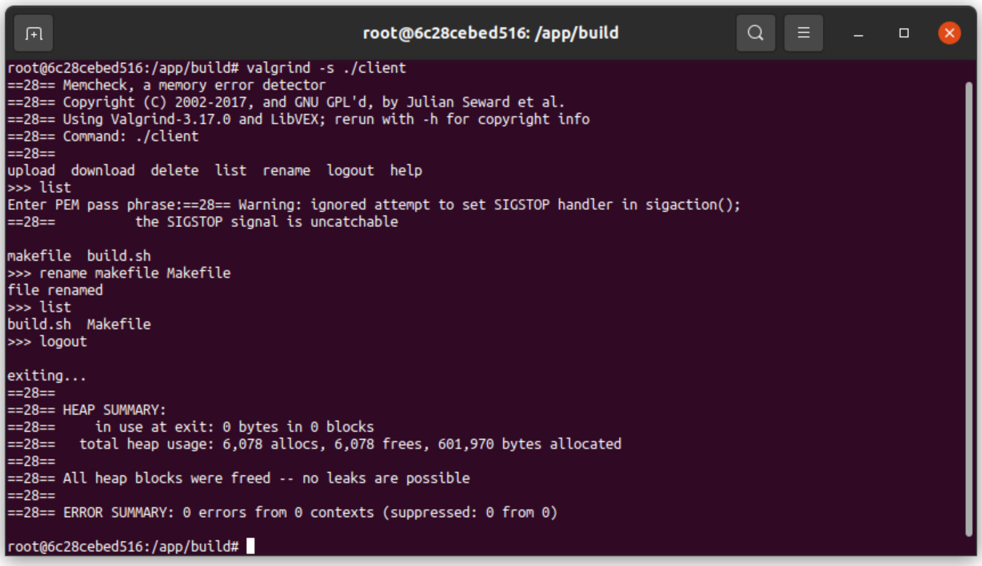
Since the user can specify the name for files that will be saved on the server, we need a way to prevent path traversal, command injection is not possible in our scenario because we never run shell commands containing user input. To prevent weird names as filenames we enforce a strict rule on filenames trough the following regex: `^\w[\w\.\-\+_]{0,19}$`. Code excerpt 5 shows the function that checks the validity of the filename.

```cpp
bool ValidPath(const std::string &path) {
  const auto re = std::regex{R"(^\w[\w\.\-\+_]{0,19}$)"};
  return std::regex_match(path, re);
}
```

Listing 5: Check for valid filenames

5 Final remarks

It's worth mentioning that using (with care [1]) C++ structures to automatically manage memory allowed us to have no memory leaks in our code (no memory errors in general), as can be seen in the output of valgrind in figure 4.



```
root@6c28cebed516:/app/build# valgrind -s ./client
==28== Memcheck, a memory error detector
==28== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==28== Using Valgrind-3.17.0 and LibVEX; rerun with -h for copyright info
==28== Command: ./client
==28==
upload  download  delete  list  rename  logout  help
>>> list
Enter PEM pass phrase:==28== Warning: ignored attempt to set SIGSTOP handler in sigaction();
==28==          the SIGSTOP signal is uncatchable

makefile  build.sh
>>> rename makefile Makefile
file renamed
>>> list
build.sh  Makefile
>>> logout

exiting...
==28==
==28== HEAP SUMMARY:
==28==     in use at exit: 0 bytes in 0 blocks
==28==   total heap usage: 6,078 allocs, 6,078 frees, 601,970 bytes allocated
==28==
==28== All heap blocks were freed -- no leaks are possible
==28==
==28== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

root@6c28cebed516:/app/build# 
```

Figure 4: Valgrind finds no memory errors in our code

[1]like avoiding reference cycles with `std::shared_ptr`