University of Amsterdam

MSc System and Network Engineering

Research Project 2

TLS Session Key Extraction from Memory on iOS Devices

Authors:
Tom Curran
tom.curran@os3.nl

Marat Nigmatullin
marat.nigmatullin@os3.nl

Supervisor:
Cedric van Bockhaven
cvanbockhaven@deloitte.nl

July 8th, 2016

Abstract

In this paper we demonstrate how the secret material used to encrypt TLS sessions can be extracted from the process memory of applications running on jailbroken and non-jailbroken iOS devices using the Frida dynamic instrumentation toolkit. This is achieved by hooking the pseudorandom function used to generate key material for TLS sessions in iOS.

We create a script to log the required material in a format ready for Wireshark to decrypt traffic. Usage of this tool is described for both jailbroken and non-jailbroken devices. Tests indicate that the tool is successful on all applications within our sample, though to varying degrees. The secret material from TLS sessions started by other processes cannot be extracted.

This work focuses on Apple's TLS implementation, CoreTLS, thus its usefulness is derived from the use of CoreTLS. With the introduction of App Transport Security in iOS 9.0, mandatory usage on the App Store due at the turn of 2017, and the relative ease with which it can be implemented in comparison to alternative TLS implementations such as OpenSSL, we anticipate the use of CoreTLS to only increase with time.

Contents

List of tables and figures

1. Introduction

Mobile devices lie at the centre of our daily lives. Beyond calls and messaging, we rely on these devices for services such as online banking, navigation, and health monitoring. As developers continue to find novel uses for these devices and the many sensors contained within, we are likely to see further integration in the future.

The unprecedented amount of information at risk - both personal and work-related – has heightened the need for secure platforms and applications. In response to this need, we see greater deployment of mobile encryption to protect this data, both at rest and in transit.

Apple's iOS is considered one of the most secure mobile operating systems and its adoption in the enterprise mobile device market, where security is a key concern, is testament to this [32]. In contrast to Android, which is open-source and made available to hardware vendors adhering to Google's specification, Apple control both hardware and software aspects of their platforms and thus limit the inherent weaknesses that arise from passing control of platform implementation to untrusted parties.

Network communications on mobile devices are predominantly secured using Transport Layer Security (TLS), successor to the Secure Sockets Layer (SSL) protocol originally developed by the Netscape Corporation in 1994. TLS provides authentication and encryption between clients and servers using a combination of public-key and symmetric-key cryptography. Key-exchange algorithms such as RSA and Elliptic Curve Diffie-Hellman (ECDHE) are used to authenticate the server – and optionally the client - and establish shared symmetric keys to encrypt the data exchanged.

Whilst the use of TLS in securing communication between two parties is good from a user perspective, it poses a necessary hurdle for security auditors to overcome when conducting black-box analysis on mobile applications. Solutions exist for circumventing TLS encryption on iOS, however they interfere with the logic of the application; either nullifying calls made to certificate validation checks or forcefully downgrading connections. Current solutions for iOS require jailbroken devices to function, potentially limiting their future use should applications deprecate support for jailbroken iOS releases or incorporate sufficient jailbreak-detection mechanisms.

In contrast to current methods for bypassing TLS, this research investigates an alternative approach by extracting the secret material used to derive the encryption keys from the memory of a process.

Problem Statement

The iOS security model focuses on providing three core elements; a secure platform, regular software updates, and tools for secure application development [22]. The success of this model can be largely attributed to its ease of use for developers and end-users. For example, the simplicity and frequency of the iOS update mechanism ensures that the majority of users install the latest updates as they are released by Apple; this is made possible by retaining control over the hardware upon which the devices execute. Looking at the distribution of iOS versions observed on the App Store in Figure 1.1, 84% of devices are running the latest major iOS release (9.0).

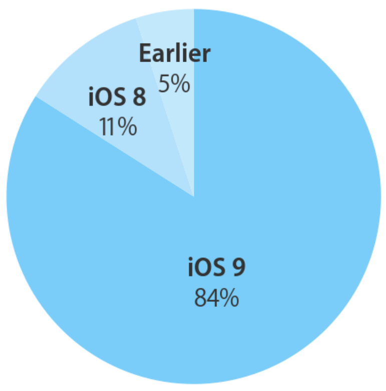


Figure 1.1: Distribution of iOS versions as measured on the App Store on May 9, 2016.

App Transport Security (ATS) is a transport security policy first introduced with iOS 9.0 designed to force encrypted connections between applications and web services. ATS achieves this by providing default connection requirements that apps and services must adhere to. ATS is automatically applied to apps compiled for iOS 9.0 and connections that do not meet these requirements will fail [2]. Presently, developers can declare exceptions to ATS to avoid server compatibility issues, however ATS support will be mandatory for all apps submitted to the App Store by the beginning of 2017 [33].

Note that ATS is not responsible for data encryption, rather promoting good security practices by enforcing strong default configurations. The encryption itself is handled by Apple's own TLS implementation which developers access via a hierarchy of high-level APIs.

Alternative TLS implementations such as OpenSSL can also be used within iOS, though it is the responsibility of the developer to compile them into their application. Apple encourage the use of their implementation over OpenSSL as it has been specifically optimised for iOS devices, reduces the size of applications, and relieves developers from having to alter their code as changes to TLS are made [8].

Whether or not it is wise for developers to rely solely on Apple for TLS is beyond the scope of this paper [1], though it will undoubtedly improve the overall deployment of encryption for apps on the App Store.

With TLS becoming a standard for apps listed on the App Store, black-box audits of iOS applications will need to bypass TLS to be able to view the traffic between client and server. Tools already exist, though they require jailbroken devices, which could be detected and are not always guaranteed. Furthermore, we anticipate the usage of Apple's TLS implementation to rise due to its ease-of-use and requirement for publishing. Taking all of the above into account, this motivates the case for a method that reliably circumvents TLS on Apple devices, regardless of whether it is jailbroken or not.

[1]Consider for example, the "go-to-fail" bug in the Apple SSL/TLS library [23].

Research Questions

It is clear there is merit in finding a new solution for overcoming TLS encryption on iOS devices. Therefore, our main research question is:

> Is it possible to extract TLS session keys from the process memory on an iOS device?

To answer this, we will address the following sub-questions:

- How is the key material used by TLS generated?
- How is TLS implemented in the latest major iOS release (9.0+)?
- How are TLS keys held in memory?
- Can such a technique be used on both jailbroken and non-jailbroken devices?

Structure

The remainder of this paper is structured as follows. In Chapter 2 we discuss existing literature relevant to our research. In Chapter 3 we review the TLS protocol and the iOS network stack. We then proceed to the problem at hand, first determining the information that needs to be extracted in Chapter 4, and then acquiring it in Chapter 5. In Chapter 6, we test this approach on a small sample of applications and compare it with existing methods. This is followed by a discussion in Chapter 7 and concluding remarks in Chapter 8. Suggestions for future work are given in Chapter 9.

2. Related Work

Research on extracting TLS/SSL session keys has been successfully conducted in the past, though similar results have yet to be realised on iOS devices. Nevertheless, as all implementations of the protocol must adhere to the original specification [20], research on other platforms is still highly relevant.

Dreijer et al. [11] successfully extracted the session ID and the master keys from the memory space of an NGINX web server running on Linux when using the `SessionTicket` extension, despite having enabled Perfect Forward Secrecy (PFS) [20]. Under PFS, it should be the case that session keys are deleted from memory immediately. During the key generation phase, an object containing the session ID and master key is allocated on the heap and is not removed until the data has been freed or the process terminated. With a time window of several minutes, it was possible to scan the process memory for the session ID and retrieve the unencrypted master key. Though possible, the authors note that it would be difficult to achieve in practice as their crawler would need to read the process' memory continuously in order obtain the session ID and key before they are removed.

TLS sessions key extraction from the memory of Android devices was demonstrated in [24]. Maritsas et al. target the OpenSSL library, commonly used in Android applications, using the Frida dynamic instrumentation toolkit [28]. Identifying the session object associated with a TLS session, they were able to extract the session ID and master secret from the default Android web browser. Their proof of concept illustrated the technique on a rooted device, though they note that it should also be possible on non-rooted devices.

As mentioned in Chapter 1, there are existing solutions to overcoming TLS/SSL encryption on jailbroken iOS devices. A prominent example is SSLKillSwitch by Diquet [9]. SSLKillSwitch is a Mobile Substrate extension that patches several functions within the Secure Transport API responsible for the certificate validation routine. Specifically, they disable Apple's built-in certificate validation routine in all SSL context objects, remove the ability to re-enable it, and then force the device to trust all custom certificates. With the device inherently trusting all certificates presented, it is possible to proxy its network traffic as it communicates to external servers. The code has changed its structure over time in response to changes in the location of the certificate validation logic, as well as its runtime behaviour with new iOS releases [10].

Another solution is to patch the binary of the application [15] so that it sends HTTP requests rather than HTTPS. If the server will only accept secure connections, a proxy to redirect traffic to and from the iOS device will be needed. In this case, the proxy will establish a secure connection to the server and relay traffic to and from the device. As with SSLKillSwitch, this method also requires a jailbroken device as modifying the application binary will invalidate its code signature (See Section 5.6.2). This technique can also fail if client validation is used on the application side or certificate pinning has been enabled.

3. Background

In this section we review the TLS protocol and the handshake mechanism used to establish a session. This is followed by an overview of the networking stack in iOS and how TLS is handled.

3.1 Transport Layer Security

Transport Layer Security (TLS) and Secure Sockets Layer (SSL) are cryptographic protocols designed to provide privacy and data integrity between two parties communicating over insecure infrastructure [20]. These protocols protect the data transferred at the transport layer in the OSI model; above TCP but below high-level protocols like HTTP.

There is a distinction between TLS and SSL. The SSL protocol was initially developed by Netscape in 1994 but never publicly released. It underwent two iterations, SSLv2.0 and SSLv3.0, the latter of which took on a completely new design to address serious weaknesses in the former [30]. In 1996, a TLS working group was formed to migrate SSL from Netscape to the Internet Engineering Task Force (IETF). TLS 1.0 was released in 1999 (RFC 2246) [19], and held only minor differences from SSLv3.0. TLS 1.1 was released in April 2006 and contained security fixes as well as introduced TLS extensions. The latest version, TLS 1.2 was released in August 2008 and added support for authenticated encryption and generally removed all hard-coded security primitives from the specification, making the protocol fully flexible.

Protocol

TLS is implemented via the *record protocol*, which handles transporting and encrypting lower-level messages exchanged over a connection. Several subprotocols are used on top of the record protocol to handle various mechanisms used during a TLS session. A TLS *record* consists of a short header containing information about its contents (e.g. subprotocol, protocol version, and length), followed by the message data itself.

The TLS specification defines four core subprotocols:

- Handshake
- ChangeCipherSepc
- ApplicationData
- Alert

The subprotocol of most interest to us is the *handshake protocol*, as it is used to negotiate the cipher suite, perform authentication, and establish a shared secret when initiating a new session between two parties. A complete protocol reference is available in the TLS 1.2 RFC [20].

Handshake

Every TLS connection begins with a handshake message and generally follows one of three flows: a full handshake with server authentication, a shortened handshake that resumes a previous session, or a full handshake with both client and server authentication.

A client and server will execute a *full handshake* and establish a *TLS session* if one has not been previously established. During this procedure, four main activities will take place:

1. Exchange capabilities and agree on best available cipher suite.
2. Validate the presented certificate(s) or authenticate using other means.

3. Agree on a shared *master secret* that will be used to protect the session.
4. Verify that the handshake messages haven't been modified by a third party.

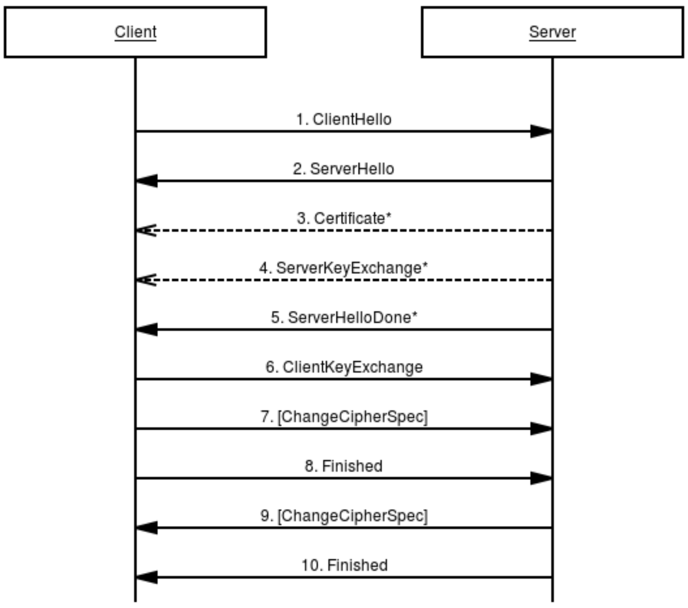


Figure 3.1: A full handshake to establish a TLS session between client and server.

Figure 3.1 lists the steps that take place in a full handshake. The client begins a new handshake by submitting its capabilities and preferences to the server with a `ClientHello` message (1). The server then selects and communicates the connection parameters, based on the best version supported by the client (2). If server authentication is required, the server will also send its X.509 certificate chain (3). If the Diffie-Hellman key exchange is selected, the server will send a `ServerKeyExchange` message containing its DH parameters. This is used in conjunction with the client's DH parameters to create a *pre-master secret* which is later used to generate the master secret from which key material is derived (4). After this, the server indicates that it has completed its side of the negotiation (5). The client sends its contribution of the key exchange (DH parameters or pre-master secret encrypted with the server's public key if using RSA) in a `ClientKeyExchange` message (6). The client now has the all information required to generate the key material and switches to encryption, informing the server using an encrypted `ChangeCipherSpec` message, a separate subprotocol from the handshake (7). The client also sends a MAC of the handshake messages sent and received (8). The server switches to encryption and informs the client (9), together with a MAC of the handshake messages it received and sent (10).

3.2 The iOS network stack

In this section we provide a brief overview of the iOS architecture and describe its networking stack.

3.2.1 iOS architecture

iOS follows a layered architecture and acts as an intermediary between applications and the underlying hardware upon which they execute. Apps communicate with the hardware via a set of defined interfaces. The lower layers of iOS contain interfaces responsible for fundamental services on the device. Higher layers build upon these lower layers to provide more sophisticated services. A graphical representation of this is depicted in Figure 3.2.

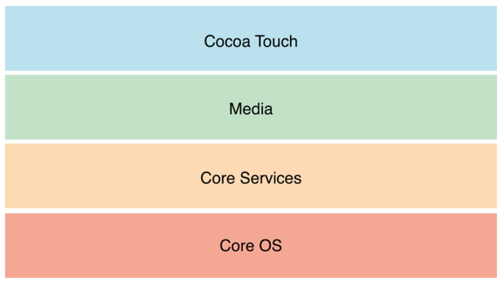


Figure 3.2: Overview of iOS architecture

Each layer of the operating system is comprised of a number of *frameworks* responsible for implementing the interfaces to specific features. A framework is a directory containing a dynamic shared library and its supporting resources such as header files, images, and helper apps [6].

The hierarchy of iOS layers is made up of four main layers: *Cocoa Touch*, *Media*, *Core Services*, and *Core OS*. The highest level of abstraction is at the Cocoa Touch layer which is responsible for implementing features such as the user interface, messaging, and map functionality and is built on top of the layers below it.

Moving down the stack reduces abstraction and provides more control over the operations performed, though at the necessary cost of greater complexity. Developers are encouraged to make extensive use of the high-level APIs made available to them in order to save time and reduce the amount of code needed to encapsulate potentially complex features such as sockets and threads [6].

The Core Services layer contains the fundamental system services for apps. Two of the most important services are the *Core Foundation* and *Foundation* frameworks which define the basic types that all apps use. This layer also contains individual technologies to support features such as location, iCloud, social media, and networking.

3.2.2 Networking APIs

The Networking APIs in iOS span several of the layers previously mentioned and are fundamentally built on top of BSD sockets, as displayed in Figure 3.3:

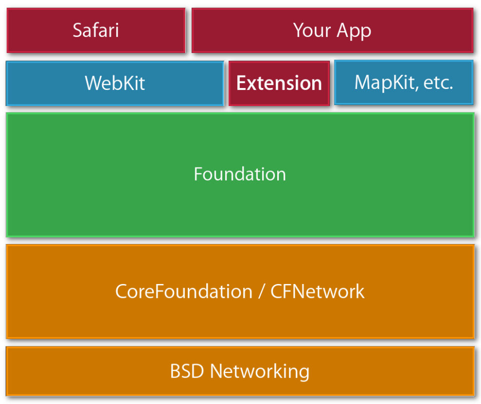


Figure 3.3: iOS network stack

At the top of the network stack is the `NSURLSession` API that is located within the Foundation framework. The `NSURLSession` API provides a method to load URL requests in order to interact with web services e.g. fetching images or videos. The `NSURLSession` API itself is built on top of the `NSStream` API which, in turn, is built on top of `CFNetwork` provided by the Core Foundation framework.

The CFNetwork Framework is a set of high-performance C-based interfaces that use object-oriented abstractions for working with network protocols. These abstractions provide detailed control over the protocol stack and make it easy to use lower-level constructs such as BSD sockets. `CFNetwork` provides more configuration options than the URL loading system offered by `NSURLSession`. For example, when creating a request using `CFNetwork`, developers must manually add any HTTP headers and cookies that must be transmitted with the request. With `NSURLSession` however, standard headers and any cookies are added automatically.

The simple networking interface offered by `CFNetwork` is in itself built on top of the `CFStream` and `CFSocket` classes. These are both lightweight wrappers around BSD sockets which sit on the lowest level in the Core OS layer, closest to the antenna hardware.

3.2.3 Security

Security in iOS is also handled at the Core OS layer by the *Security* Framework. This provides interfaces for security-related operations such as managing certificates, public and private keys, and trust policies. Within the Security Framework is the `Secure Transport` API, which is responsible for transparently setting up TLS sessions with remote services when invoked.

The stack of networking APIs in iOS and their dependence across different layers in iOS demonstrates that Apple-provided network security, at their lowest-level, relies upon the Secure Transport API. These relations are presented in Figure 3.4.

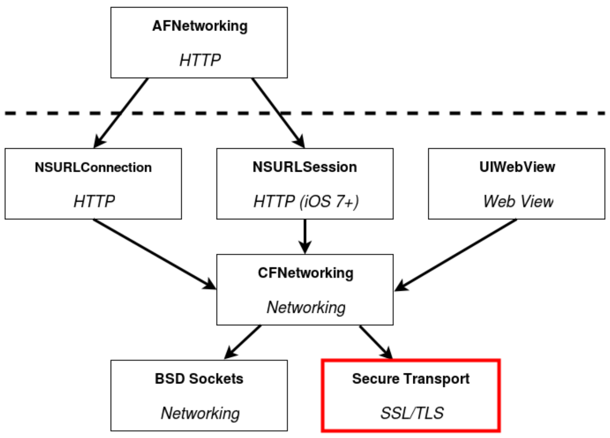


Figure 3.4: Relationship between iOS networking APIs

3.2.4 App Transport Security

App Transport Security (ATS) is a new feature introduced with iOS 9.0 that enforces default connection requirements to ensure that apps adhere to best practices for secure connections when using high-level APIs such as `NSURLConnection`, `NSURLSession`, and `CFURL`. With ATS enabled, network connections that do not meet the specified requirements will fail.

Servers must support a minimum of TLS 1.2, forward secrecy, and certificates must be valid and signed using SHA-256 or better with a minimum of a 2048-bit RSA key or 256-bit elliptic curve key. Apple specify a list of requirements including the supported cipher suites, which generally require the use of Elliptic Curve Diffie-Helmann with Ephemeral keys (ECDHE) for the key exchange and either Elliptic Curve Digital Signature Algorithm (ECDSA) or RSA for authentication [2].

ATS is enabled by default on applications built against the iOS 9 and OS X 10.11 SDKs, and will be mandatory for *all* applications on the App Store by the beginning of 2017. It is currently possible to define exceptions for ATS-enabled applications for servers that do not support the required ciphers, though it is possible that Apple will start enforcing its use in the future.

With the introduction and enforcement of ATS on iOS applications, usage of Apple's Secure Transport API will undoubtedly increase in the future. Furthermore, for application developers whose main concern is delivering core functionality within tight deadlines, it is likely that the majority of TLS encryption on iOS devices going forward will be handled by Apple's own TLS implementation.

4. Approach

In this section we determine what must be retrieved in order to decrypt TLS traffic. We begin by outlining how the key material is generated according to the TLS specification. We then identify the information that will be extracted later in Chapter 5. After this, we turn our focus to Apple's TLS implementation where we analyse source code snippets to determine which objects and functions contain the required information.

4.1 Generation of key material

TLS makes extensive use of an HMAC-based Pseudorandom function (PRF) in order to expand secrets into blocks of data that can be used for key generation and validation. The PRF function takes as input a *secret*, a *seed*, and an ASCII *label* to produce an output of arbitrary length. As the process is deterministic, both client and server can start with the same seed values and PRF function to arrive at the same final values.

The PRF is defined in the TLS RFC [20] as:

```
PRF(secret, label, seed) = P_<hash>(secret, label + seed)
```

where:

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                       HMAC_hash(secret, A(2) + seed) +
                       HMAC_hash(secret, A(3) + seed) + ...
```

and A() is defined as:

```
A(0) = seed
A(i) = HMAC_hash(secret, A(i-1))
```

The `P_hash` function can be iterated as many times as necessary in order to generate the required amount of material and will depend on the hashing algorithm used. For example, using SHA-256 will produce 256 bits (or 32 bytes) of output with each iteration. Thus in order to generate 48 bytes in total, the function will be iterated twice, and the remaining 16 bytes discarded. Recall that by the end of the handshake, both parties will possess a shared secret (`pre_master_secret`) and the random values sent in the `ClientHello` and `ServerHello` messages, respectively. Both the client and server random values act as a source of entropy in the generation of the 48-byte master secret as they are unique to each TLS session. The master secret is generated using these values in conjunction with the PRF function:

```
master_secret = PRF(pre_master_secret, "master secret" + client_random +
↪  server_random)
```

Listing 1: Master secret generation

The master secret is then used to generate the `key block` which contains the MAC secrets, symmetric encryption keys, and input vectors for blocking used with symmetric encryption. These are, in order, the client write MAC secret, server write MAC secret, client write key, server write key, client write IV, and server write IV.

```
key_block = PRF(SecurityParameters.master_secret,
                "key expansion",
                SecurityParameters.server_random +
                SecurityParameters.client_random);
```

Listing 2: Key material generation

Once enough output has been generated, the `key_block` is partitioned according to the respective lengths for each of the keys like so:

```
client_write_MAC_key[SecurityParameters.mac_key_length]
server_write_MAC_key[SecurityParameters.mac_key_length]
client_write_key[SecurityParameters.enc_key_length]
server_write_key[SecurityParameters.enc_key_length]
client_write_IV[SecurityParameters.fixed_iv_length]
server_write_IV[SecurityParameters.fixed_iv_length]
```

Listing 3: Partitioning the key block

What is needed

Our goal is retrieve enough information such that we can derive the key material used to encrypt a given TLS session. As discussed, the key material is generated using a PRF function in conjunction with the random values exchanged at the start of a handshake and the shared secret computed at each endpoint. The relationships between these values are displayed in Figure 4.1.

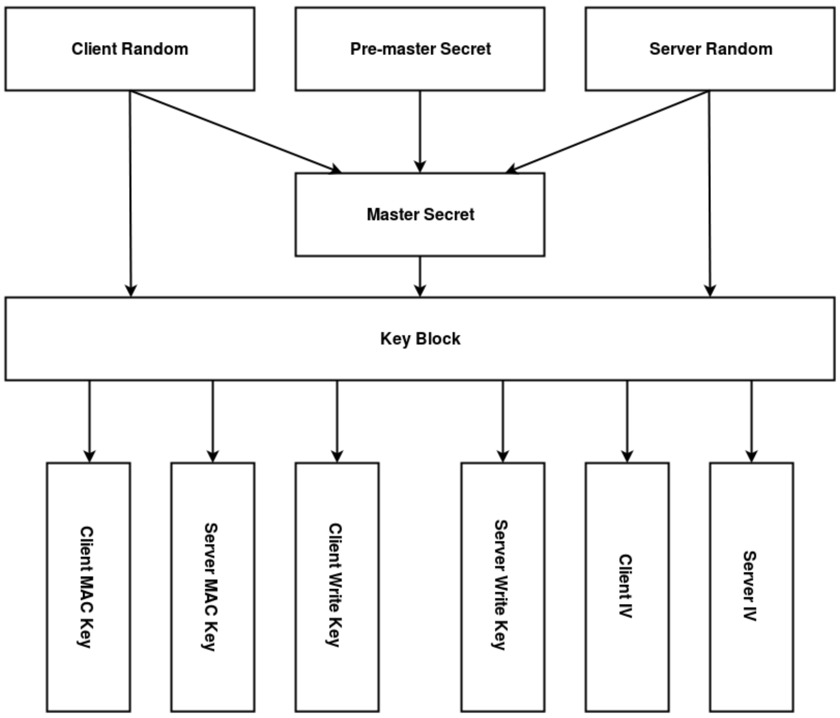


Figure 4.1: Relationship between secret and key material.

In order to generate the key block, we need both random values as well as the master secret or pre-master secret. At a minimum the random values and the pre-master secret are required as these will be used to generate the master secret. Note that both the client and server random values are sent in plaintext, and are thus available to anyone observing the handshake.

4.2 Taking a closer look at Secure Transport

As mentioned, the high-level networking APIs available in iOS rely on the Secure Transport API for handling TLS sessions. Upon inspection of the source code of the Secure Transport API it became apparent that this in itself wasn't responsible for the low level cryptographic operations and was actually a wrapper.

Articles concerning previously disclosed SSL vulnerabilities stated that Apple patched their own SSL library, *CoreTLS* [25]. With knowledge of the library, we were able to find the source code, albeit for an older version than what is currently in use [3]. Our next steps were to analyse the source to locate objects containing the random values and master secret, followed by functions that handle these objects so they could be later hooked.

Analysis of the CoreTLS Source

The structure definition of the `_tls_handshake_s` object contains all information related to the session and is our main target. A snippet from this definition is shown in Listing 4.

```
struct  _tls_handshake_s {

        tls_protocol_version  negProtocolVersion;
            tls_protocol_version  clientReqProtocol;
            tls_protocol_version  minProtocolVersion;
            tls_protocol_version  maxProtocolVersion;
            ...

        uint8_t             clientRandom[SSL_CLIENT_SRVR_RAND_SIZE];
        uint8_t             serverRandom[SSL_CLIENT_SRVR_RAND_SIZE];
        tls_buffer                  preMasterSecret;
        uint8_t             masterSecret[SSL_MASTER_SECRET_SIZE];
        ...
}
```

Listing 4: Snippet of `tls_handshake_s structure`

The structure begins with four entries containing TLS protocol versions. These are used when parties negotiate which cipher suite to use during a handshake. Later entries contain the client random, server random, pre-master and master secrets which can be used to generate the key material.

Whenever a new session is to be established, an instance of an object that has a `_tls_handshake_s` structure will be created, as shown in the snippet below (Listing 5):

```
typedef struct _tls_handshake_s *tls_handshake_
```

Listing 5: Creation of the instance

Our next step was to determine the function(s) that interact with this object and leads us to the `tls_handshake_internal_prf` function. This is the PRF function used to generate the key material. Listing 6 shows the function definition:

```
int tls_handshake_internal_prf(tls_handshake_t ctx,
                               const void *vsecret,
                               size_t secretLen,
                               const void *label,
                               size_t labelLen,
                               const void *seed,
                               size_t seedLen,
                               void *vout,
                               size_t outLen);
```

Listing 6: `tls_handshake_internal_prf` function definition

The first argument is the object containing the session information, whilst the second argument is a pointer to the master secret stored *within* that object. The third argument contains the length of the master secret, which should be 48-bytes as defined in the RFC. The fourth and fifth arguments contain a pointer to the static label and its length, respectively. This is followed by the seed values.

Additional Information

In addition to the secret material, more information about the session can be gleaned from other functions contained within the CoreTLS library and Secure Transport API. For example, the cipher suite being used for the connection can be extracted from the `tls_record_init_pending_ciphers` function as well as the URLs corresponding to each TLS session by hooking the `SecPolicyCreateSSL` function from the Secure Transport API. Whilst not crucial to decrypting the network traffic, these metrics could still provide some use (e.g. metrics when analysing large quantities of applications).

With knowledge of the object containing the random values and master secret, as well as its internal structure, combined with the function that handles this object, we can now retrieve them using dynamic instrumentation tools as described in the next chapter.

5. Extracting the Secret Material

In the previous section we established that in order to generate the key material we need the master secret and the two random values exchanged at the start of a TLS session. In this section we describe how this information can be retrieved. First we provide an overview of the tools we used, followed by the retrieval itself. The process is described for both jailbroken and non-jailbroken devices.

5.1 Test setup

The following test environment was used:

- MacBook Pro (OS X 10.11.5 “El Capitan”)
- iPad Mini 3 (iOS 9.0.2, Jailbroken)
- iPhone 5C (iOS 9.0.2, Jailbroken)
- iPhone 5 (iOS 9.3.2, Non-Jailbroken)

Our test devices include both jailbroken and non-jailbroken devices, as well as different processor architectures. The iPad Mini 3 uses Apple’s A7 64-bit system on a chip (SoC) which has an ARMv8-A instruction set. Both the iPhone 5 and iPhone 5C use an A6 32-bit SoC with ARMv7-A instruction set.

5.2 Tools

Wireshark

Wireshark was used to capture and decrypt the network traffic sent and received by the iOS devices [12]. It includes an SSL/TLS dissector which is able to decrypt traffic if provided with the 32 byte random value from the `ClientHello` message and the master secret. The random value is used to map the master secret to its corresponding TLS session. In order to do this, it must be passed a text file that holds the keys in the format shown in Listing 7:

```
CLIENT_RANDOM <32 byte client random value> <48 byte master secret>
```

Listing 7: File structure required by Wireshark SSL/TLS decryption.

Frida

Frida [28] is dynamic instrumentation toolkit that facilitates the live inspection of processes without access to the source code and in a manner that leaves the process unaware. It achieves this by spawning a dedicated thread for a Frida agent within the targeted process. A client is able to communicate with this agent remotely. A JavaScript (JS) runtime is loaded into the process, allowing the client to send custom scripts for instrumentation.

Once running, Frida has access to anything within the context of the process. For example, one can inspect and modify memory, threads, and registers. It can also be used to enumerate threads, hook arbitrary functions, and call native APIs. There is a lot of functionality built into Frida, a more complete list of the JavaScript API is available on the project website [29].

It currently supports all versions of iOS, using the Duktape JS engine [27] and works on both jailbroken and non-jailbroken devices. Frida scripts operate in the same manner on jailbroken and non-jailbroken devices, they only differ in how the agent is loaded on the device.

To operate on jailbroken devices, a Frida package must be installed via Cydia [13]. This will install the frida-server application that runs as a daemon on the device and is responsible for loading the agent library (FridaGadget.dylib) into the targeted process.

The process for non-jailbroken device differs as the frida-server application is unavailable. Instead, the agent must be manually injected into the target application, a process that is described in Section 5.6.

On the client-side, the frida-trace is a command-line tool that can be used to trace arbitrary functions. When tracing a function for the first time it generates a JS handler within which instrumentation code can be inserted. This code will be executed when hooking the function, either on entry or exit.

5.3 Obtaining the Master Secret

Recall that in Chapter 4 we determined that the secret material could be extracted by hooking the tls_handshake_internal_prf function. In order to do this, we first run the frida-trace tool on the tls_handshake_internal_prf and read the first 400 bytes of the session object passed as its first parameter. The output is displayed in 9.

We can confirm that we have the correct object by observing the output from the first 16 bytes. This corresponds to the hard-coded values for TLS 1.2, as displayed in tls_types.h from the CoreTLS source code (See Listing 8). We see 0303 repeated four times, stored in little-endian format. This data corresponds to the protocol negotiation described in Listing 4

The master secret can be retrieved by reading the memory from the pointer passed as the second argument to tls_handshake_internal_prf. As the master secret is stored within the main object, this can also be seen in Listing 9 from offsets 0x00000148-0x00000177.

```
    tls_protocol_version_SSL_3 = 0x0300,
    tls_protocol_version_TLS_1_0 = 0x0301,
    tls_protocol_version_TLS_1_1 = 0x0302,
    tls_protocol_version_TLS_1_2 = 0x0303,
    tls_protocol_version_DTLS_1_0 = 0xfeff,
```

Listing 8: Hardcoded values for protocol versions in tls_types.h

```
tls_handshake_internal_prf()
- offset -    0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F  0123456789ABCDEF
0x00000000  03 03 00 00 03 03 00 00 03 03 00 00 03 03 00 00  ................
0x00000010  00 00 00 00 c0 31 b2 3b 00 00 00 00 00 00 00 00  .....1.;........
0x00000020  00 00 00 00 00 00 00 00 40 7d f4 17 00 00 00 00  ........@}......
0x00000030  00 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x00000040  00 00 00 00 d0 de f4 17 03 00 00 00 00 00 00 00  ................
0x00000050  00 00 00 00 ff ff ff ff 00 00 00 00 00 00 00 00  ................
0x00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x00000070  00 00 01 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x00000080  00 00 00 00 00 00 00 00 00 00 00 00 47 00 00 00  ............G...
0x00000090  a0 42 f1 17 31 10 00 00 00 4e dd 18 01 00 00 00  .B..1....N......
0x000000a0  0e 00 00 00 30 f6 f4 17 0e 00 00 00 30 88 03 19  ....0.......0...
0x000000b0  01 00 00 00 00 01 00 00 30 c0 00 00 30 c0 00 00  ........0...0...
0x000000c0  11 00 00 00 20 0c 0c 00 04 00 00 00 50 99 e9 17  .... .......P...
0x000000d0  30 00 00 00 00 00 00 00 00 00 00 00 0f 00 00 00  0...............
0x000000e0  00 00 00 00 1b 00 00 00 b0 fa 00 19 00 00 00 00  ................
0x000000f0  02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x00000100  57 72 87 62 1f ab 7f 15 c0 41 c8 41 dd 2e 52 18  Wr.b.....A.A..R.
0x00000110  51 fa 6d a4 7a ff fa 5c c8 cf 6a 16 25 d8 02 68  Q.m.z..\..j.%..h
0x00000120  57 72 87 62 a8 07 3b 57 26 2c 50 99 7a 69 f7 7e  Wr.b..;W&,P.zi.~
0x00000130  ea 2c bc 28 bd 18 9d 85 96 de 9b 65 14 0d aa 27  .,.(.......e...'
0x00000140  00 00 00 00 00 00 00 00 3f f4 37 1f 35 a5 9d d0  ........?.7.5...
0x00000150  70 04 98 f6 e0 58 d4 2b 00 d3 74 d6 49 d1 a6 cf  p....X.+..t.I...
0x00000160  6d ae 3f 96 b4 06 65 4e 09 9f 72 64 56 2c b3 27  m.?...eN..rdV,.'
0x00000170  d9 68 e0 5e 4b 36 04 f3 60 00 00 00 60 9c f3 17  .h.^K6..`...`...
0x00000180  5c 00 00 00 80 76 f4 17 6c 00 00 00 e0 25 f1 17  \....v..l....%..
```

Listing 9: `tls_handshake_internal_prf` function object dump

5.4 Obtaining the Client and Server Random Values

There are several options to retrieve the client and server random values. For example, the random values are included in the session object, and thus they can be found in the dump displayed in Listing 9. All that must be determined is the specific offsets at which the data is stored and these can be obtained by reading the object data and cross-referencing the structure with the corresponding Wireshark traffic. A drawback to this approach however is that the offsets will change when using devices with different architectures (e.g. 32-bit vs 64-bit).

A more pragmatic approach is to use the other arguments passed to `tls_handshake_internal_-prf`. Recall that the PRF function defined in Chapter 4 is passed a static label concatenated with a seed. The seed itself is a concatenation of both random values.We can read from the `seed` pointer (6th parameter) for `seedLen` bytes (7th argument), which should be 64 bytes, and then split this value into two 32-byte values, we will have both random values. This approach yields a more optimal solution than the latter as it will correctly retrieve the values regardless of device architecture.

The results of the second operation is shown below in Figure 10:

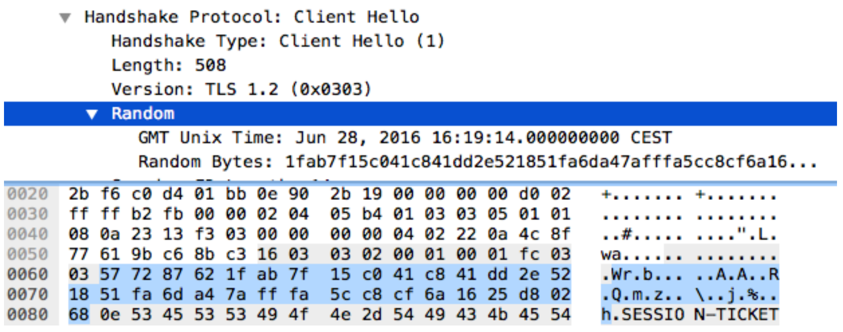
```
tls_handshake_internal_prf()
Client_random: 577287621fab7f15c041c841dd2e521851fa6da47afffa5cc8cf6a1625d80268
Server_random: 57728762a8073b57262c50997a69f77eea2cbc28bd189d8596de9b65140daa27
```

Listing 10: Client and server random values as extracted from `seed` argument

5.5 Verification of secrets

To verify the extracted values, we first compare our values with Wireshark and then attempt to decrypt traffic with the retrieved values. We can confirm that we have the correct random value even without knowledge of the shared secret as the `ClientHello` and `ServerHello` packets are transmitted in plaintext.

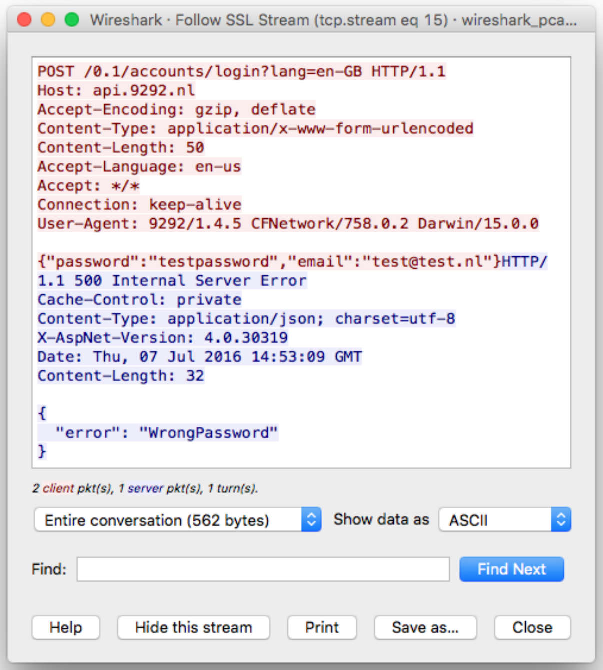
A visual comparison of the the client random value displayed in Wireshark with the retrieved value from Listing 9 is shown in Figure 5.1 and confirms that we have the correct values. We next check the master secret by creating a text file using the format described earlier in Listing 7.



```
▼ Handshake Protocol: Client Hello
    Handshake Type: Client Hello (1)
    Length: 508
    Version: TLS 1.2 (0x0303)
  ▼ Random
      GMT Unix Time: Jun 28, 2016 16:19:14.000000000 CEST
      Random Bytes: 1fab7f15c041c841dd2e521851fa6da47afffa5cc8cf6a16...
0020  2b f6 c0 d4 01 bb 0e 90  2b 19 00 00 00 00 d0 02   +....... +.......
0030  ff ff b2 fb 00 00 02 04  05 b4 01 03 03 05 01 01   ........ ........
0040  08 0a 23 13 f3 03 00 00  00 00 04 02 22 0a 4c 8f   ..#..... ....".L.
0050  77 61 9b c6 8b c3 16 03  03 02 00 01 00 01 fc 03   wa...... ........
0060  03 57 72 87 62 1f ab 7f  15 c0 41 c8 41 dd 2e 52   .Wr.b... ..A.A..R
0070  18 51 fa 6d a4 7a ff fa  5c c8 cf 6a 16 25 d8 02   .Q.m.z.. \..j.%..
0080  68 0e 53 45 53 53 49 4f  4e 2d 54 49 43 4b 45 54   h.SESSIO N-TICKET
```

Figure 5.1: Client random value as displayed in Wireshark

After passing this file to Wireshark's SSL/TLS dissector, we are able to successfully view the decrypted traffic for the TLS session (see Figure 5.2).



Wireshark · Follow SSL Stream (tcp.stream eq 15) · wireshark_pca...

```
POST /0.1/accounts/login?lang=en-GB HTTP/1.1
Host: api.9292.nl
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
Content-Length: 50
Accept-Language: en-us
Accept: */*
Connection: keep-alive
User-Agent: 9292/1.4.5 CFNetwork/758.0.2 Darwin/15.0.0

{"password":"testpassword","email":"test@test.nl"}HTTP/
1.1 500 Internal Server Error
Cache-Control: private
Content-Type: application/json; charset=utf-8
X-AspNet-Version: 4.0.30319
Date: Thu, 07 Jul 2016 14:53:09 GMT
Content-Length: 32

{
  "error": "WrongPassword"
}
```

2 client pkt(s), 1 server pkt(s), 1 turn(s).

Entire conversation (562 bytes) Show data as ASCII

Find:

Figure 5.2: Decrypted TLS stream in Wireshark

5.6 Non-Jailbroken devices

A significant benefit from extracting TLS key material on a non-jailbroken device is that an application is unaware that it's taking place. Developers often incorporate both certificate pinning and jailbreak-detection mechanisms into their applications - applications could halt their execution or behave differently according to the environment they believe to be running in. By extracting the keys on a non-rooted device and allowing the TLS session to take place under normal conditions, we are able to observe the application's true behaviour.

The process of obtaining the secrets on non-jailbroken devices is similar to jailbroken devices, they differ only in how the Frida agent (`FridaGadget.dylib`) is loaded into the target application. In this section we describe how to insert the Frida agent and resign the application so it will launch successfully on a non-jailbroken device.

5.6.1 Inserting the Frida gadget

Applications on iOS are packaged as IPA (`.ipa`) archives which contain the binaries and supporting resources (e.g. frameworks, libraries, images). Despite the file extension, they are simply renamed ZIP archives and can be treated as such.

They are structured as follows [34]:

- iTunesArtwork
- iTunesMetadata.plist
- Payload/
  - {ApplicationName}.app/
    - _CodeSignature/
    - Frameworks
    - Various application files e.g. app binary, images, libraries.

To insert the Frida gadget, we simply copy the `FridaGadget.dylib` into the `Payload/ApplicationName.app/Frameworks/` directory. We then insert a load instruction for the Frida library in the application binary so that it's loaded when the application is launched.

Application binaries on iOS are encrypted to prevent examination, but decrypted once loaded into device memory (See Listing 11). To decrypt the binary the `Clutch` tool can be used [7]. This tool will first inspect the binary, to determine the offset (`cryptoff`) and size of the encrypted data (`cryptsize`) and then launch the application. Once launched, `Clutch` will dump the decrypted data and use it to patch over the encrypted portion of the binary before removing the `cryptid` flag. To run `Clutch` however requires a jailbroken device.

```
$ otool -arch all -Vl 9292 | grep -A5 LC_ENCRYP
          cmd LC_ENCRYPTION_INFO
      cmdsize 20
     cryptoff 16384
    cryptsize 3194880
      cryptid 1
Load command 13
```

Listing 11: Example of an encrypted iOS binary

Without access to the source code of the application, the binary cannot be rebuilt to include a load instruction for the Frida gadget. Instead we can use `optool` to manually insert a load command for the Frida library [35].

5.6.2 Code-signing on iOS

To verify the authenticity of executable code running on the device, iOS requires native code to be signed by a known and trusted certificate. This protection is referred to as *Mandatory Code Signing* (MCS). There are several components to Apple's code signing security model:

- Developer Certificates
- Provisioning Profiles
- Signed Applications
- Entitlements

Developer certificates can be obtained through Apple's iOS Developer Program and are required in order to run custom applications on an iOS device and submit them to the App Store [1].

A Provisioning Profile is an XML plist file signed by Apple that configures the iOS device to permit the execution of code signed by the embedded developer certificate. It also lists the entitlements that the developer is permitted to grant to applications holding their signature. A Provisioning Profile can be acquired through either the iOS Developer Program or Xcode (since version 6) [16].

All iOS executable binaries and applications must be signed by a trusted certificate. While Apple's certificates are inherently trusted, any other certificates must be installed via a properly signed Provisioning Profile. The signature for a signed executable is embedded inside the Mach-O binary file format, or in the extended file system attributes if it's a non-Mach-O executable such as a shell script. This way, any executable binary can be signed: dynamic libraries, command line tools, and `.app` bundles [21].

Executables and can be signed and verified by the `codesign` tool and the developer's certificates. When an application bundle is signed, a `CodeResources` file will be created in the `_CodeSignature` directory. This file is used to store the signatures of all signed files in the bundle.

In addition to verifying an application against its signing certificate, the Entitlements of an application are checked against the certificate to determine its access to the system resources in the iOS sandbox. In other words, code signing is used to ensure application integrity, whilst entitlements restrict what the application actually has access to. Entitlements are specified as an XML file generated by Xcode when configuring the capabilities of an application. This file is provided to the `codesign` tool to embed them in the signature of the application.

The signatures of an application are checked when an application is installed, as well as at runtime. An initial check is performed to prevent the installation of potentially malicious unsigned apps. Dynamic code signing checks of all executable memory pages are made as they are loaded at runtime to ensure that an app has not been modified since it was installed or last updated, this is known as *Code Signing Enforcement* (CSE) [2]. CSE is built into the iOS kernel's virtual memory system.

The code signing validity of a process is tracked with the `CS_VALID` flag in the `csflags` member of the kernel's `proc` structure for that process. If the executable's code signing signature has been validated prior to it being executed, the process begins execution with the `CS_VALID` flag set. If the process becomes invalid, this flag will be cleared. The `CS_KILL` flag is set by default on iOS and will forcibly kill a process once it becomes invalid [36].

5.6.3 Loading an re-signed application on a non-jailbroken device

The procedure for launching a resigned application also differs between jailbroken and non-jailbroken devices. Resigning the application with a valid certificate meets one condition – allowing the installation of the application on a non-jailbroken device. However, the function hooking implemented by Frida necessarily replaces part of the application code during runtime.

The jailbreak process applies several kernel patches on the device, one of which targets the *Apple Mobile File Integrity* (AMFI) kernel module, preventing it from setting the `CS_HARD` and `CS_KILL` flags inside the process' code-signing flags. As these flags *are* set on non-jailbroken devices however, resigned applications must be launched using a different method.

Currently there are a few methods available to launch resigned applications, though the simplest is to launch the application with the `lldb` debugger attached – this is the approach taken in this paper. When `lldb` is attached, the kernel disables `CS_KILL` and `CS_HARD` for the process, thereby disabling code-signing.

5.6.4 Summary

To summarise, the process of preparing and launching an application for instrumentation with Frida on non-jailbroken devices are:

1. Download target app from the App Store on a jailbroken device.
2. Use `Clutch` to retrieve the decrypted IPA file.
3. Insert the Frida Gadget and use `optool` to add a load instruction on the binary.
4. Acquire provisioning profile or developer certificate and install onto test device.
5. Re-sign the application.
6. Deploy and launch the application with a debugger attached (e.g. using `ios-deploy`).

5.7 Automation

To aid our research we created two scripts: one to automate the extraction with Frida, and another to resign the application when preparing an app for non-jailbroken devices.

Key extraction was handled using Frida's Python API and supports jailbroken and non-jailbroken devices. After connecting the device via USB and starting a Wireshark capture on its network traffic, one need only start the script together with the bundle identifier for the application to be tested. The script will spawn the application on the device and automatically begin logging the client random and master secret values to a text file. Upon finishing the packet capture, this file can be passed to Wireshark to decrypt the traffic.

The script used to resign the applications was created by Grezia [17], which uses the `codesign` tools that under code-signing in Xcode. It requires the use of a Mac device.

An example of the tools terminal output and the generated secrets file is presented in the Appendix B.2.

6. Evaluation

In this section we review the performance of our method on a sample of applications from the App Store. We then compare the advantages and disadvantages between this method and existing methods to circumvent TLS encryption.

6.1 Decrypting application traffic

We ran the tool with five applications available on the App Store. We began capturing Wireshark traffic before starting the application with the script and generating some arbitrary traffic. The results of these tests are displayed in Table 6.1:

Application	Version	Category	Developer	Decryption
9292	1.4.4	Travel	Reisinformatiegroep B.V.	Full
Facebook	59.0	Social Networking	Facebook, Inc.	Partial
Instagram	8.4	Photo & Video	Instagram, Inc.	Partial
My UvA	1.0.8	Education	University of Amsterdam	Full
Snapchat	9.43.3.0	Photo & Video	Snapchat, Inc.	Partial

Table 6.1: Test results from key extraction using Frida

Secret material was successfully extracted from all applications, though to varying degrees. Specifically, we possessed secrets to all sessions established in the My UvA and 9292 applications however this wasn't the case for the other three. We were unable to find the exact reason behind this, though we offer two possible explanations.

Many of the sessions we do not have the material for are *resumed* TLS sessions. That is, they are reusing secrets from previously established sessions with new random values. This is evident from the fact that a `session ID` value is sent across in several of the `ClientHello` packets initially sent when spawning the application. Session resumption allows for a shorter TLS handshake and is used to reduce the overhead when establishing a session. The iOS networking APIs such as `NSURLConnection` and `NSURLSession` do maintain TLS session caches, though presumably they are removed once a process is terminated [4]. Oddly, when running our tool on non-jailbroken devices, we necessarily install the application on the device before running it. Thus, when the application is running on the device and its traffic captured, it is for the first time.

In addition to resumed sessions, there are also sessions established outside of the process we hook. We know this from observing the GeoIP information from the packets sent. Present in all applications, we see connections to Apple servers based in Cupertino. As these sessions are established outside of the process we are targeting, we are unable to extract the relevant secret material.

6.2 Comparison with existing solutions

We compare our method with two existing methods, SSLKillSwitch and manually patching the binary, on the basis of four aspects:

1. Requirements

2. Level of decryption
3. TLS session interaction
4. Shortcomings

Requirements

All three methods require the use of a jailbroken device at *some* point. SSLKillSwitch requires root privileges in order to apply its patches to disable all certificate validation. Manually patching an application binary will break code signatures and will thus fail to launch unless code signature checks have been disabled. Using Frida, we require a jailbroken device insofar as to obtain a decrypted application binary to prepare for instrumentation - a prepared binary will run on an unmodified device.

Level of decryption

We define the level of decryption in terms of how much of the total traffic on the device is decrypted. As SSLKillSwitch is used in conjunction with a proxy that intercepts all traffic and allows disables certificate validation, all traffic on the device is decrypted. As one would expect, manually patching a binary will yield unencrypted traffic for only the target application. The same is the case when using Frida, and this is because we are only able to hook calls made to the TLS PRF function from the context of the process we are instrumenting. As observed during our tests, the key material for sessions established by other processes will not be obtained.

TLS session interaction

Interaction is defined as pro-actively influencing how a TLS session is established. Of the three methods, the TLS session is left untouched only when using Frida, and this is due to the approach. When binary patching or using SSLKillSwitch, the device connects to a proxy and not its intended endpoint. In absence of the key material, this is a requirement in order to view unencrypted communications. One of the key advantages to extracting the key material is the ability to observe an applications behaviour under normal conditions.

Shortcomings

We evaluate shortcomings on the basis of environmental factors that could cause a given solution to stop working. Influencing a TLS session creates two points of failure for SSLKillSwitch and binary patching. Firstly, a jailbroken device is needed in order to load the modifications - this can be thwarted with sufficient jailbreak-detection mechanisms deployed in applications. Secondly if client authentication is used by the server, as is often the case with Mobile Device Management (MDM) apps, both solutions will always fail as the device does not communicate directly with the external server. The method described in this paper is unaffected by both of these problems as it can run on an unmodified iOS device and passively extracts the required key material. Frida is currently able to avoid anti-debugging measures, though this could change in the future.

Aspect	**SSLKillSwitch**	**Binary patching**	**Hook TLS PRF function with Frida**
Requirements	• Jailbroken device to install Cydia tweak	• Jailbroken device to launch modified binary	• Jailbroken device to obtain decrypted binary
Level of decryption	• All traffic on device	• App-specific	• App-specific
TLS session interaction	• Device communicates with proxy using self-generated certificate	• Device communicates via HTTP or proxy	• None
Shortcomings	• Jailbreak detection • Unknown certificate pinning method • Client authentication by server	• Jailbreak detection • Certificate pinning • Client authentication by server	• Anti-debugging measures prevent Frida from instrumenting application

Table 6.2: Comparison with alternative approaches

7. Discussion

We were able to decrypt traffic in the previous chapters by hooking calls made to the PRF function used to generate the master secret. This approach is advantageous over alternatives that rely on continuously scanning and parsing the memory as discussed in Chapter 2 as the master secret is extracted *as* it is generated. Therefore it is unaffected by this factors such as PFS which significantly reduce the window of time before the key is removed from memory. Of course by specifically targeting the CoreTLS library we do however introduce dependencies, both on its implementation as well as the TLS protocol specification itself.

A new version of TLS, version 1.3, is currently a working draft [18] however we do not anticipate this will affect the efficacy of our method. For the most part, TLS 1.3 proposes reducing the number of cipher suites in use to prevent the use of weak or broken ciphers such as MD5 and SHA-224 [26]. TLS 1.3 also introduces "*the 1 round trip time*" (1-RTT) handshake, which aims to reduce the number of packets sent to establish a connection. This is achieved by requiring the client to send the `ClientKeyExchange` message, containing the client's cryptographic parameters, *before* a cipher suite has been negotiated. In this way, the server is then able to calculate the keys for encryption and authentication before the first packet is sent. As we target the PRF function used to generate the secret material in TLS and thus should be unaffected by this change.

In addition to its implementation, our work also depends on the usage of Apple's TLS library. However, as mentioned in Chapter 1, the introduction of App Transport Security (ATS) and the relative ease with which developers can incorporate the Secure Transport API into their applications is likely to increase the use of CoreTLS over time. Though it is possible to use alternative TLS implementations such as OpenSSL, Apple does not advise this. Notably in [8], developers are encouraged to use the Secure Transport API, stating their encryption libraries have been optimised for iOS hardware, security patches will be implemented by Apple, and it will reduce the total size of the applications. Ultimately, this promises developers switching to Secure Transport faster applications and a smaller footprint without the time spent manually updating their application code to support updates to TLS.

We used Frida to dynamically instrument the application and thus our solution is also dependent on the success of this tool. However one of Frida's aims is to avoid traditional anti-debugging measures. Though one could invest time in isolating the functionality necessary to extract secret material and removing this dependency, time may be better spent working on improving Frida given its applicability to multiple platforms.

A jailbroken device is needed in order to acquire the decrypted application to instrument on the non-jailbroken device. We also used a machine running OS X in order to perform the app resigning. However, this could be resolved on non-Mac platforms using tools such as `libmobiledevice` [31] and `ldid` [14].

8. Conclusion

In this paper we have demonstrated how the secret material used to encrypt TLS sessions can be dynamically extracted from the process memory of applications running on jailbroken and non-jailbroken iOS devices. This was achieved using the Frida dynamic instrumentation toolkit.

After reviewing the relationships between iOS networking APIs, we identified a central dependency on the Secure Transport API when securing network connections. Analysis of the source code for the CoreTLS library used by the Secure Transport API revealed the implementation of the PRF function used to generate the key material and this was subsequently hooked to retrieve the secret material as it was generated using Frida.

Using Frida's Python bindings, we created a short script that will spawn an arbitrary application and begin logging the required secret material to a file that can be passed to Wireshark to decrypt traffic. The use of this tool was described for both jailbroken and non-jailbroken devices.

Tests indicate that the tool is successful on all applications within our sample, though to varying degrees. Some applications, notably those from companies such as Facebook, Snapchat, Instagram, contain multiple layers of encryption which prevent us from accessing everything that is sent over the wire. On the other hand, the traffic from smaller applications such as My UvA and 9292 are fully decrypted.

The tool has been specifically built for Apple's TLS implementation, CoreTLS, thus its usefulness is derived from the use of CoreTLS. With the introduction of ATS in iOS 9.0, mandatory usage due at the turn of 2017, and the relative ease with which it can be implemented in comparison to third party TLS implementations, we anticipate the use of CoreTLS to only increase with time.

9. Future Work

Our tool is currently able to extract TLS secret material from applications that rely on Secure Transport. Despite this, there are still several areas that warrant future research.

We believe it would be beneficial to investigate the traffic we could not obtain the key material for. Firstly, by extending the tool to simultaneously handle multiple services on a non-jailbroken device. This would expose the traffic generated in the case of subsidiary applications helping the main. In addition we would like to look more closely into how TLS sessions are cached on the device. In particular we noticed with the Facebook application that a TLS session was immediately resumed, despite never having installed or launched the application on the device. This could of course be due to human error, which we would like to verify.

Another area of future work would be to test this tool against future versions of iOS such as 10 which was just recently released. As mentioned in Chapter 7, our tool targets on the CoreTLS library and is therefore sensitive to changes to it. As such, we expect the tool to work on future releases of iOS, provided that the CoreTLS library remains unchanged.

In future we would like to also extend the tool to support third-party TLS implementations such as OpenSSL. Over the course of our research we downloaded and tested over 50 applications from the App Store in an attempt to hook functions used by OpenSSL however they were not in use. Indeed it could be the case that developers have simply stopped using it [1], perhaps a more focused effort can find alternative implementations from which to extract secret material.

With access to the shared secrets established between a client and a server it may be possible to incorporate this into a man-in-the-middle setup (MITM). For example, by resuming an existing session between client and server and simultaneously connecting to them using the same key material. The MITM would be able to decrypt incoming traffic from the client, inspect or modify it (and the HMAC), and then pass it onto the server.

[1]Some developers have stopped maintaining OpenSSL frameworks for iOS on GitHub, encouraging users to switch to using the Secure Transport API [5].

Bibliography

[1] Apple. *Certificates - Support - Apple Developer.* June 13, 2016. URL: https://developer.apple.com/support/certificates/.

[2] Apple. *iOS Security (iOS 9.3 or later).* June 8, 2016. URL: https://www.apple.com/business/docs/iOS_Security_Guide.pdf.

[3] Apple. *Source Browser - coreTLS source code.* June 6, 2016. URL: https://opensource.apple.com/source/coreTLS/coreTLS-83.20.8/lib/.

[4] Apple. *Technical Q&A QA1727: TLS Session Cache.* June 17, 2016. URL: https://developer.apple.com/library/ios//qa/qa1727/_index.html.

[5] Stefan Arentz. *ios-openssl.* June 7, 2016. URL: https://github.com/st3fan/ios-openssl.

[6] Jack Cox, Nathan Jones, and John Szumski. *Professional iOS network programming: connecting the enterprise to the iPhone and iPad.* John Wiley & Sons, 2012.

[7] Kim Jong Cracks. *Clutch: Fast iOS executable dumper.* June 14, 2016. URL: https://github.com/KJCracks/Clutch.

[8] Paul Danbold. *Security and Privacy in iOS 7.* June 7, 2016. URL: https://developer.apple.com/videos/play/enterprise/17.

[9] Alban Diquet. *SSL Kill Switch 2 - Blackbox tool to disable SSL certificate validation - including certificate pinning - within iOS and OS X Apps.* June 3, 2016. URL: https://github.com/nabla-c0d3/ssl-kill-switch2.

[10] Alban Diquet. *SSL Kill Switch and Twitter iOS.* June 3, 2016. URL: https://nabla-c0d3.github.io/blog/2016/02/21/ssl-kill-switch-twitter/.

[11] Joey Dreijer and Sean Rijs. *Perfect forward not so secrecy.* UVA, SNE, May 31, 2016.

[12] Wireshark Foundation. *Wireshark - Go Deep.* June 16, 2016. URL: https://www.wireshark.org/.

[13] Jay Freeman. *Cydia.* June 7, 2016. URL: https://cydia.saurik.com/.

[14] Jay Freeman(saurik). *ldid git repository.* June 20, 2016. URL: http://gitweb.saurik.com/ldid.git.

[15] Prateek Gianchandani. *IOS Application Security Part 28 – Patching IOS Application with Hopper.* June 7, 2016. URL: http://resources.infosecinstitute.com/ios-application-security-part-28-patching-ios-application-hopper.

[16] Jay Graves. *What is a Provisioning Profile? (Pt. 2) by Jay Graves of Double Encore.* June 13, 2016. URL: https://possiblemobile.com/2013/04/what-is-a-provisioning-profile-part-2/.

[17] Giovanni Di Grezia. *Resign your iOS ipa (Frameworks and Plugins included).* June 17, 2016. URL: http://www.xgiovio.com/blog-photos-videos-other/blog/resign-your-ios-ipa-frameworks-and-plugins-included/.

[18] IETF. *draft-ietf-tls-tls13-14 - The Transport Layer Security (TLS) Protocol Version 1.3.* June 7, 2016. URL: https://tools.ietf.org/html/draft-ietf-tls-tls13-14.

[19] IETF. *RFC 2246 - The TLS Protocol Version 1.0.* June 6, 2016. URL: https://tools.ietf.org/html/rfc2246.

[20] IETF. *RFC 5246 - The Transport Layer Security (TLS) Protocol Version 1.2.* June 6, 2016. URL: https://tools.ietf.org/html/rfc5246.

[21] Thomas Kollbach. *Inside Code Signing.* June 14, 2016. URL: https://www.objc.io/issues/17-security/inside-code-signing/.

[22] Ivan Krstic. *How iOS Security Really Works.* June 7, 2016. URL: https://developer.apple.com/videos/play/wwdc2016/705/.

[23] Adam Langley. *Apple's SSL/TLS bug (22 Feb 2014).* June 7, 2016. URL: https://www.imperialviolet.org/2014/02/22/applebug.html.

[24] Stamatios Maritsas, Yadvir Singh, and Kenneth van Rijsbergen. *SSN Project. SSL session key extraction from the process memory of a running Android application.* UVA, SNE, May 30, 2016.

[25] Charlie Osborne. *Apple patches dozens of security flaws in iOS 8.4, OS X 10.10.4.* June 6, 2016. URL: `http://www.zdnet.com/article/apple-patches-dozens-of-security-flaws-in-ios-8-4-os-x-10-10-4/`.

[26] Sean Parkinson. *Secure Crypto: TLS 1.3 – A New Beginning - Speaking of Security - The RSA Blog and Podcast.* June 7, 2016. URL: `https://blogs.rsa.com/secure-crypto-tls-1-3-new-beginning/`.

[27] Ole André Vadla Ravnås. *Frida 7.2 Released.* June 7, 2016. URL: `http://www.frida.re/news/2016/06/02/frida-7-2-released/`.

[28] Ole André Vadla Ravnås. *Frida • A world-class dynamic instrumentation framework.* June 7, 2016. URL: `http://www.frida.re/`.

[29] Ole André Vadla Ravnås. *JavaScript API -Frida • A world-class dynamic instrumentation framework.* June 7, 2016. URL: `http://www.frida.re/docs/javascript-api/`.

[30] Ivan Ristic. *Bulletproof SSL and TLS: Understanding and Deploying SSL/TLS and PKI to Secure Servers and Web Applications.* Feisty Duck, 2013.

[31] Martin Szulecki. *libimobiledevice - A cross-platform software library and tools to communicate with iOS devices natively.* June 6, 2016. URL: `http://www.libimobiledevice.org/`.

[32] Good Technology. *Mobility Index Report Q1 2015.* June 7, 2016. URL: `https://www.scribd.com/doc/264890478/Good-Mobility-Index-Report-Q1-2015`.

[33] techparse. *Apple mandates App Store apps support ATS security protocol by 2017.* June 8, 2016. URL: `http://techparse.co.uk/2016/06/15/apple-mandates-app-store-apps-support-ats-security-protocol-by-2017/`.

[34] The iPhone Wiki. *IPA File Format - The iPhone Wiki.* June 17, 2016. URL: `https://www.theiphonewiki.com/wiki/IPA_File_Format`.

[35] Alex Zielenski. *optool: Command Line Tool for interacting with MachO binaries on OSX/iOS.* June 17, 2016. URL: `https://github.com/alexzielenski/optool`.

[36] Dino A. Dai Zovi. *Apple iOS Security Evaluation.* June 17, 2016. URL: `http://securitylearn.net/wp-content/uploads/iOS%20Resources/Apple%20iOS%204%20Security%20Evaluation%20WP.pdf`.

Acknowledgements

We would like to thank Cedric van Bockhaven at Deloitte for his support and guidance throughout the duration of this project. We would also like to thank the developers of Frida, whose tool was *instrumental* in helping us achieve our goals.

Appendices

A. Source Code

A.1 Frida script to retrieve client and server random values and master secret

```
#!/usr/bin/python

import frida
import sys


def on_message(message, data):
    try:
        if message:
            f = open("wireshark.txt",'a')
            print("{0}".format(message["payload"]))
            f.write(message["payload"]+"\n")
    except Exception as e:
        print(message)
        print(e)


hook = """
var f = Module.findExportByName("libsystem_coretls.dylib",
↪  "tls_handshake_internal_prf");
var client_old;
var master_old;
var premaster_old;
var clientrandom="";
var serverrandom="";
var ctr = 0;
Interceptor.attach(f, {onEnter: function (args) {
                  var masterlength = parseInt(args[2],16);
                console.log("masterlength: " + masterlength);
            var seedlength = parseInt(args[6],16);
                console.log("seedlength: " + seedlength);

                if (seedlength == 64)
                        {
                        randoms = new NativePointer(args[5]);
                        readb = Memory.readByteArray(randoms,seedlength)
                        uintreadb = new Uint8Array(readb)
                        array = [];
                        for (i=0;i<32;i++)
                          {
                              array[i] = ("0"+
↪  uintreadb[i].toString(16)).substr(-2);
                        clientrandom = clientrandom + array[i];
                               }
                        for (i=32;i<64;i++)
                              {
```

```
                        array[i] = ("0"+ uintreadb[i].toString(16)).substr(-2);
                        serverrandom = serverrandom + array[i];
                           }
                }

                if(masterlength ==48)
                           {
                               var masterstring = "";
                             var premasterstring ="";
                                  masterarg=args[1];
                                  var master = Memory.readByteArray (masterarg
 ↪  ,masterlength);
                               tempconvert = new Uint8Array(master);
                               arr = [];
                       for ( i = 0; i < masterlength; i ++)
                                {
                          arr [ i ] = ("0" + tempconvert [ i ]. toString (16)).
 ↪  substr ( -2);
                          masterstring = masterstring + arr[i];
                                }
                          if (masterstring != master_old)
                                    {
                           master_old = masterstring;
                                client_old = clientrandom;
                                send("CLIENT_RANDOM "+ clientrandom + " "+
 ↪  masterstring);

                                ctr++;
                                console.log("Current counter = " + ctr)
                                        console.log("Client random: " +
 ↪  clientrandom)

                                       console.log("Server random: " +
 ↪  serverrandom)

                                        clientrandom = "";
                                        serverrandom = "";
                                    }
                        }
}
});
        """
if __name__ == '__main__':
    try:
        print("[*] Attaching ...")
        print("[*] Hooking... \n")
        if sys.argv[1] == "-n":
                session = frida.get_usb_device().spawn(["re.frida.Gadget"])
        else:
                session = frida.get_usb_device().spawn([sys.argv[1]])
        session1 = frida.get_usb_device().attach(session)
        frida.get_usb_device().resume(session)
        script = session1.create_script(hook)
        script.on('message', on_message)
        script.load()
        sys.stdin.read()
    except KeyboardInterrupt:
        sys.exit(0)
    except IndexError:
        print("""
Error: No process specified.
Usage: dumper.py [BundleIdentifier]
E.g. dumper.py com.apple.mobilesafari
If device is jailbroken please use -n option
""")
```

A.2 Frida script to retrieve session URL

```
onEnter: function (log, args, state) {
        log("SecPolicyCreateSSL(" + "" + ")");
        a=args[1].toInt32()+8;
        var url = new NativePointer(a);
        url = Memory.readByteArray (url ,1);
        var urllength = new Uint8Array(url);
        finallength = urllength[0].toString();
        urllocation = a+1;
        urllocation = new NativePointer(urllocation)
        finallength = parseInt(finallength,16); //conver to hex
        var urlv = Memory.readCString(urllocation, finallength);
        log("URL: " + urlv);
        log("URL length: " + finallength);
    },
```

A.3 Frida script to retrieve hexadecimal value of the cipher used

```
onEnter: function (log, args, state) {
        log("tls_record_init_pending_ciphers(" + "" + ")");
        var cipher = args[1];
        log(cipher);
    },
```

A.4 Frida script to retrieve hexadecimal value of the minimum protocol version

```
onEnter: function (log, args, state) {
        log("tls_handshake_set_min_protocol_version(" + "" + ")");

        log(args[1]);
    },
```

A.5 Frida script to retrieve hexadecimal value of the max protocol version

```
onEnter: function (log, args, state) {
        log("tls_handshake_set_max_protocol_version(" + "" + ")");

        log(args[1]);
    },
```

B. Example output

B.1 Script

```
$ ./dump.py nl.9292.9292
[*] Attaching ...
[*] Hooking...

CLIENT_RANDOM 5788e98c4673d61dc4bf065bc36ee2ff5cc6917ab0a9464816a3049e3881b21c
 ↪  502185b9e167f88f84ffcdbaea60c54df2bbff6e56e51026430549dff363f3acd5851276...
Current counter = 1
Client random: 5788e98c4673d61dc4bf065bc36ee2ff5cc6917ab0a9464816a3049e3881b21c
Server random: 5788e98dfd5be28540fe52e3f28be6e2829215c638ab8750fac8002bfde3f3aa
CLIENT_RANDOM 5788e98e438ea427fd3df55d29754639122586e8a375eb76e29d82d8b1646b87
 ↪  606258ebe8019aae808f820698b0ef83ac8e6a8e60f139ff8f2d045cde1a9bc5c4ef450e8...
Current counter = 2
Client random: 5788e98e438ea427fd3df55d29754639122586e8a375eb76e29d82d8b1646b87
Server random: 5788e98eacdd48039c50ea3b9f8a97b66e565c57477fab2671fdce392f4b54bf
CLIENT_RANDOM 5788e98e53590c3a00fdd3b11ef8de3b72fcc697b834dfd050be4440b5054cda
 ↪  f888ab70c18c06f9b11d23dc45a791583b51e4485e6b36ae82ee18dd0d8e185b6788e82a5...
Current counter = 3
Client random: 5788e98e53590c3a00fdd3b11ef8de3b72fcc697b834dfd050be4440b5054cda
Server random: 5788e98e00c7721586a832e26430fadeade06bb34ce41366bb9275b5f74ae3f5
CLIENT_RANDOM 5788e9b8ed3759739c4d1c7eb47d0708de931477281d603842a3d8aa5d55687b
 ↪  487d83713c89a2cb445691c87e8095e53a1af666ebe5d97b204e45026fcee28e131d731e0...
Current counter = 4
Client random: 5788e9b8ed3759739c4d1c7eb47d0708de931477281d603842a3d8aa5d55687b
Server random: 5788e9b8e16fbcba0d2f09464e8bdd23cd81af4e26028e76b3f26e044b9a6dc0
CLIENT_RANDOM 5788e9b98b7e7f006a3385c1bf9f4a889638aed3be9229118fe81e87e9ef5018
 ↪  e78e714254548fc9e8db40958441ee5ac722dd705ef1f0cd504d745aae0bacc96ea5a1ee0...
Current counter = 5
Client random: 5788e9b98b7e7f006a3385c1bf9f4a889638aed3be9229118fe81e87e9ef5018
Server random: 5788e9b9625f703e9c143b6f92b4471460bcdd90d35c7c1d8819918f6760d0f9
```

B.2 Secrets file

```
CLIENT_RANDOM 5788e98c4673d61dc4bf065bc36ee2ff5cc6917ab0a9464816a3049e3881b21c
 ↪  502185b9e167f88f84ffcdbaea60c54df2bbff6e56e51026430549dff363f3acd5851276...
CLIENT_RANDOM 5788e98e438ea427fd3df55d29754639122586e8a375eb76e29d82d8b1646b87
 ↪  606258ebe8019aae808f820698b0ef83ac8e6a8e60f139ff8f2d045cde1a9bc5c4ef450e...
CLIENT_RANDOM 5788e98e53590c3a00fdd3b11ef8de3b72fcc697b834dfd050be4440b5054cda
 ↪  f888ab70c18c06f9b11d23dc45a791583b51e4485e6b36ae82ee18dd0d8e185b6788e82a...
CLIENT_RANDOM 5788e9b8ed3759739c4d1c7eb47d0708de931477281d603842a3d8aa5d55687b
 ↪  487d83713c89a2cb445691c87e8095e53a1af666ebe5d97b204e45026fcee28e131d731e...
CLIENT_RANDOM 5788e9b98b7e7f006a3385c1bf9f4a889638aed3be9229118fe81e87e9ef5018
 ↪  e78e714254548fc9e8db40958441ee5ac722dd705ef1f0cd504d745aae0bacc96ea5a1ee...
```