Abstract

Client platform infection poses a significant threat to secure user authentication. Combining vulnerable client platforms with special security devices, as often the case in e-banking, can increase significantly the security. This paper describes a new architecture where a security proxy on the client platform communicates with both a trusted security device and the server application. The proxy switches between two TLS channels, one from the client and another from the trusted device. The result is a highly usable and flexible authentication solution with strong security assurance.
## Contents

1 Introduction 4
   1.1 Motivation .................................................. 4
   1.2 Research questions ............................................ 4
   1.3 Our contributions ............................................. 4

2 Security Architectures 5
   2.1 Trusted Proxy Architecture ................................. 5
   2.2 Trusted Offline Device Architecture ......................... 5
   2.3 Offline Trusted Device Proxy Architecture .................... 6

3 TLS Switching 7
   3.1 Background of TLS ............................................. 7
   3.2 The TLS Switching concept .................................... 7
   3.3 Requirements and Security analysis .......................... 8

4 TLS Switching Implementation 10
   4.1 Naive implementation based on serialization .................. 11
   4.2 Optimized implementation ..................................... 11
   4.3 Evaluation ..................................................... 11
   4.4 Discussion about implementations ............................. 12

5 Proxy Architecture 13
   5.1 Proxy communication ........................................... 13
   5.2 Filters and commands .......................................... 14

6 Applications of TLS Switching 15
   6.1 Receiving sensitive information ............................... 15
   6.2 Sending sensitive information ................................ 15
      6.2.1 Detecting a specific token ............................... 16
      6.2.2 Identifying client messages ............................. 16
      6.2.3 Identifying server messages ............................. 17
   6.3 Trusted device and server authentications .................... 17
   6.4 TLS Switching based protocols ............................... 18

7 User and account management 18
   7.1 Profiles ........................................................ 18
   7.2 Backup, export and import .................................... 19
   7.3 Sharing profiles and company management ..................... 19

8 Usability 19
   8.1 User authentication to the trusted device ..................... 20
   8.2 User interaction ................................................ 20
   8.3 User behaviour ................................................. 21

9 Trusted device used for the OTDP architecture 22
CONTENTS

10 Related Work
   10.1 TLS Handshake Proxying ........................................ 22
   10.2 TLS Splitting ................................................... 22

11 Future Work ......................................................... 23

12 Discussion and Conclusion ......................................... 23
1 Introduction

1.1 Motivation

Online services typically require user authentication based on user credentials, before granting access to the service. Thus, only the registered, authorized and authenticated users are supposed to access the services. However, attackers compromising the client terminal can enable them to obtain the credentials of a user, and to falsely authenticate and be granted access to the services, and possibly misuse the user identity in other domains as well.

Solutions to protect against compromised client platforms are typically based on secure external devices, but have limits. Indeed, this would required a specific configuration on the server-side or putting the secure external device between the client and the server. By using a proxy, that the client must install in order to access the Internet, and that is able to send requests to a secure external device, it is possible to handle all authentication protocols without exposing the credentials in the client-side. Moreover, the external device can also be assumed to communicate only with the proxy, thus limiting its attack surface.

The secure external device can establish a TLS connection with the server through the proxy in order to send credentials. Thanks to TLS Handshake, the proxy can use the same connection to send and receive data from, or for the client without allowing the client to know the credentials send by the secure external device.

The secure external device can provide additional features, such as whitelist, blacklist and certificate validation, in order to authenticate the server, and can easily be extended to add other security features.

The user is the main vulnerability of a holistic system, so it is important to ensure security usability in order to assist in security cognition and decision making. A security proxy can provide security usability through advanced security ergonomics design, in order to correctly interpret and handle server responses, and to signal when to use the secure features of the external device.

In case multiple users are using the same device, or the same user uses it in different contexts, the secure external device also provides specific user profile management. This makes the user able to authenticate in a secure and user-friendly way, where the protection applies not only to the server side or to the client-server connection, but also to the client and the user’s side.

1.2 Research questions

In this technical report, we aim to exchange sensitive informations with a server through an untrusted client. Our work focus on user-side security, usability, ergonomics and transparency problematics.

1.3 Our contributions

By combining two existing client-server architectures involving an external trusted device, we created a new architecture we name Trusted Device Proxy Architecture. As OTDP is transparent both to client and server applications, OTDP adoption choice is left to final users, if not required by applications providers. We are intimately convinced this represent a major strength for OTDP users’ adoption. A first version of OTDP has been implemented without support of TLS.
We invented and implemented *TLS Switching*, allowing to redirect a TLS connection end while being transparent to the other end. TLS Switching is used by OTDP to allow a trusted device to exchange sensitive data with a server through a client able to read or write on the same TLS connection, while remaining transparent to the server. TLS Switching is not to be confused with TLS handshake proxying.

Finally, we give some applications of OTDP with TLS Switching along with some recommendations.

## 2 Security Architectures

We assume in a traditional client-server architecture that the client can be infected with malware, e.g., certificates could be replaced with untrusted ones to facilitate phishing attacks [20] or a keylogger [27] could be installed to intercept user inputs like credentials. The server is assumed to be secured from such attacks.

For applications with higher levels of security requirements, such as online banking, it is common to use an additional external trusted device [19]. This can be a transportable device pluggable into the client or a secure part of the client s.a., a secure element, a Trusted Platform Module [6], or a Trusted Execution Environment\(^1\) like the Intel’s SGX [3].

### 2.1 Trusted Proxy Architecture

A trusted device can be placed between the client application and the server, as in Figure 1. We call this a trusted proxy, with examples including the SSL proxy concept [13] or the Bitdefender BOX\(^2\) in Internet of Things. The trusted proxy is transparent to both the client and the server, and is assumed secure. All data passes through the trusted proxy, allowing it to read, log, modify, or suppress the data.

Thus the trusted proxy could be used to perform actions without the user’s approval. Moreover, encrypting and decrypting data in the trusted proxy increases the latency of the connection and can overload the capacity of small trusted devices. More importantly for us is that such trusted proxies could lead to a false feeling of security and data non-repudiation.

### 2.2 Trusted Offline Device Architecture

To avoid passing all data through the trusted device, the client may communicate directly with the server, and only query the trusted device for sensitive information, as in Figure 2.

---

\(^{1}\)http://www.globalplatform.org/specificationsdevice.asp

\(^{2}\)http://www.bitdefender.com/box/
Communication from the trusted device to the server goes through the (possibly infected) client. Therefore, any sensitive information would need to be protected.

The trusted device may be invisible to the server, e.g., in applications using zero-knowledge [15] or challenge-response [8] protocols, in which case the client would use the trusted device for doing computations on sensitive information that should not be exposed on the client side. This method is used by HTTP eXtended Digest Access Authentication (HTTP XDAA) [18], an HTTP DAA [12] version in which authentication is relegated to a trusted device.

Often the trusted device is known to the server and used in a specific algorithm designed requiring a trusted device. Banking applications are such examples [2], where the trusted device holds a secret unavailable on client application, for example a private key certified by the trusted device provider.

In other algorithms the trusted device is just an extra security feature, e.g., MP-Auth [23] encrypts a password with the server’s public key. The algorithm uses a trusted device, but does not require one: the client application is able to respond to the server without having a trusted service.

The client, on the other hand, must be programmed to use the trusted device, usually in form of plugins for each client application. The HTTP XDAA and MP-Auth both use browser plugins, and these are not compliant with other browsers. Writing specific code for each client application that is likely to be used with this architecture is costly, proportional to the number of client applications to adapt and the number of protocols used for communications between the client application and trusted service.

2.3 Offline Trusted Device Proxy Architecture

We propose an architecture that combines the two above, the Offline Trusted Device Proxy (OTDP) architecture, where a proxy is sitting on the client side, as shown in Figure 3, and communicates with an offline trusted device.

The client communicates with the server through the proxy, as was the case in the first architecture. Therefore, this architecture is transparent for both the client and the server, but
still allows to read, log, modify, and suppress exchanged data. The proxy is assumed to be open, so the user can read the sources, build it from the sources, modify or replace it. Thus spying on the user or doing actions without the user’s approval is harder for the provider. Moreover, there is no false feeling of security since the proxy is also considered untrusted as it runs on the client platform.

As in the trusted offline device architecture, sensitive information are processed by the offline trusted device upon requests from the proxy. To protect this information we develop the concept of TLS Switching, presented in Section 3, which will not require specific algorithms or protocols on neither the client nor the server. We only need to instrument the client platform by installing the proxy.

3 TLS Switching

3.1 Background of TLS

TLS is a secure communication protocol standardized by the Internet Engineering Task Force (IETF) [9] providing confidentiality and data integrity.

TLS is a two-layer protocol having as basis the Record protocol, in which TLS records carry messages of types Application Data, Handshake, Change Cipher Spec and Alert. The TLS record protocol ensures confidentiality and integrity based on symmetric key encryption algorithms which use keys established in the Handshake protocols.

For the TLS Switching concept we focus on the Handshake protocol, leaving out the Change Cipher Spec and Alert protocols. Application messages are then exchanged in the Application Data protocol that uses the keys given after a successful handshake.

In the first handshake message CLIENT HELLO, the client proposes a set of cipher suites to use and a set of supported TLS extensions [7]. The server then selects a cipher suite and TLS extensions, and notifies its decision to the client in a SERVER HELLO. The handshake protocol then negotiates a premaster secret used to generate symmetric keys and initialization vectors for the selected cipher suite.

The Handshake protocol provides forward and backward secrecy as older secrets cannot be deduced from newer ones, nor the other way. Handshakes are done regularly, e.g., to renew keys and initialization vectors, with subsequent runs of the Handshake protocol being called renegotiations. The initial handshake is performed unencrypted.

To reduce the connection latency between the client and a server, the client might establish a TLS connection to an edge server. To avoid giving the server’s private key to the edge server, the initial TLS handshake is performed by the edge server querying the server for operation involving the private key [26]. Such operation is called TLS Handshake Proxying.

Please note that the 2009 attacks on TLS renegotiation is fixed, since 2010, by the "renegociation_info" TLS extension [10].

3.2 The TLS Switching concept

In the OTDP architecture we use two channels of communication with the server: one from the proxy (channel P) and one from the trusted service through the proxy (channel T). Since the proxy resides on the untrusted platform of the client it must be unable to read or write
on channel T. As the OTDP architecture should be compliant with traditional client-server architectures, the channels should be seen on the server-side as one (see Figure 4).

The TLS Switching works in two modes (or states of operation):

**In state T** the channel T is used to exchange sensitive information with the server.

**In state P** the channel P is used to transmit, in a more efficient way, information between the server and the client applications.

![Figure 4: TLS Switching](image)

In both states the TLS handshake and record protocols are used. Initially the connection is in state T.

**State P** is reached when the trusted service renews the keys and initialization vectors with a renegotiation, and gives the new keys and initialization vectors to the proxy. The proxy is then able to read and write records through a standard TLS connection.

**State T** is reached again when the trusted service performs a renegotiation to renew the keys and initialization vectors. Due to forward and backward secrecy properties, the proxy is not able to deduce keys and initializations vectors from the ones it knows. Thus the proxy is not able to read or write records during the state T, i.e. to read or write records on the channel T.

From the client point of view the TLS Switching is transparent, and it operates as shown in figure 5. A secure connection preexists between the trusted service and the proxy (0). The client establishes a TLS connection over TCP (1) with the proxy seen as the server. The proxy then establishes a TCP connection with the server (2a) over which the trusted service establishes a TLS connection (3). After eventually exchanging informations with the server, the current connection go into state P (4) allowing the client to communicate with the server (5). Before the TLS connection setup, the proxy can send information to the trusted service such as the server’s domain name or the TLS extensions used by the client (2b).

### 3.3 Requirements and Security analysis

TLS Switching requires:

- the server to accept TLS renegotiations;
- the proxy to support the cipher suite and TLS extensions used in state P
- the trusted service to verify TLS certificates
the trusted service and the proxy must be able to exchange the current cipher suite identifier, symmetric keys and initialization vectors in a secure way.

As the OTDP architecture cannot be differed from the standard client-server architecture by the server, the proxy runs on the client, and we assume the proxy-trusted service communication secure, there is no security fallback compared to the standard architecture.

Since the client is untrusted, channel P is untrusted as well. Therefore, sensitive information must be exchanged on channel T during state T.

In state T, the sensitive information exchanged between the trusted service and the server, is protected from the client platform by the forward and backward secrecy of the TLS handshake. In state T a corrupted proxy cannot enforce state P as it cannot read or write records to perform a renegotiation. In order to avoid phishing attacks, server authentication during handshake must be required and performed by the trusted service before exchanging sensitive information.

Non-repudiation and trusted service authentication can be achieved by performing client authentication during handshake and the state needs to be known to the server. This can be achieved by either expecting different certificates for different states or accepting an empty certificate for state P. Such features should be declared in the CLIENT HELLO's TLS extensions.
4 TLS Switching Implementation

TLS is implemented by various libraries such as GnuTLS\(^3\) or OpenSSL\(^4\) in C, and JSSE\(^5\) in Java. A possible pseudo-code implementation of TLS Switching is proposed in Listing 1. The proxy should only process application data messages (line 11).

To be able to perform a renegotiation in state P, the trusted service needs to know, from the proxy, the internal state of the current cipher engines (lines 14, 18, 19).

Listing 1: TLS Switching pseudo-code

```java
void TrustedService::goStateP() {
    handshake();
    SessionInfo sessionInfo = extract();
    sendToProxy(sessionInfo);
}

void Proxy::talkWithServer() {
    SessionInfo sessionInfo = receive();
    setSessionInfo(sessionInfo);
    while (!trustedServiceWantToTalk && receiveTLSAppDataRecord) {
        // talk with server
    }
    sendToTrustedService(sessionInfo);
}

void TrustedService::goStateT() {
    SessionInfo sessionInfo = receive();
    setSessionInfo(sessionInfo);
    handshake();
}
```

In the context of this paper, a TLS Switching proof of concept was implemented in Java:

- **Java version**: 1.7.0_95
- **OpenJDK Runtime Environment**: IcedTea 2.6.4 7u95-2.6.4-1-deb8u1
- **OpenJDK Client VM**: build 24.95-b01, mixed mode, sharing

The proxy and the trusted service exchange TLS records over TCP. In state T, the proxy forwards the trusted service’s messages to the server and the server’s messages to the trusted service. Two types of messages are added:

- `'t`': messages used to go into state T.
- `'p`': messages used to go into state P.

---

\(^3\)http://www.gnutls.org/

\(^4\)https://www.openssl.org/

\(^5\)http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136007.html
4 TLS SWITCHING IMPLEMENTATION

4.1 Naive implementation based on serialization

In a first naive implementation, we used TLS Switching without worrying about the performances.

In this implementation, the $p$ and $t$ messages both carry a serialized SSLEngineImpl allowing to build and read records on the proxy. The $p$ and $t$ messages have a record header, and their content is clear for the proxy to read it. To go into state $P$, the trusted service performs a renegotiation and then sends a $p$ message to the proxy. To go into state $T$, the proxy sends a $t$ message to the trusted service which then performs a renegotiation.

As the proxy is considered untrusted, we improved this naive implementation by selecting the SSLEngineImpl fields to serialize in the $p$ and $t$ messages. Then we partially build a SSLEngineImpl which can now be used to read or write records, but unable to perform renegotiations.

To return into state $T$, as the SSLEngineImpl internal state is modified after reading and writing records, the trusted service’s SSLEngineImpl has to be updated through $t$ messages before performing a renegotiation.

4.2 Optimized implementation

We can optimize $p$ and $t$ messages by sending the minimum amount of required information.

The $p$ messages are now used to build the SSLEngineImpl’s fields which were serialized in the naive version. For that we send through $p$ messages the selected cipher suite identifier, the TLS protocol version, and the asymmetric keys.

The $t$ messages are now used to update the internal state of the SSLEngineImpl which only requires two sequence numbers, incremented for each record sent or received under the current keys.

The $p$ messages do not have to contain sequence numbers as they can be deduced by the proxy. Moreover, $p$ and $t$ messages do not have to contain initialization vectors. Indeed, RC4 cipher suites are prohibited in TLS 1.2 [25] thus cipher suites only offer block ciphers (AES with CBC or 3DES with CBC) or NULL cipher. Yet, in block ciphered messages, initialization vectors are included in the record’s message and are randomly chosen for each record.

4.3 Evaluation

To test the TLS Switching proof of concept, three virtual machines were created in VirtualBox, called here: "Proxy", "Trusted" and "Server". The server runs an OpenSSL server (version 1.0.1k 8 Jan 2015) with 2048 bits RSA keys.

The tests are conclusive and prove that both TLS Switching implementations are functional.

Table 1 gives the cost in bytes of each operation. The costs of going into state $P$ or $T$ do not include the cost of the required renegotiation.

Initially the naive implementation costed near 26.4kB and was reduced to 13.4kB by removing null values and SunJCE (com.sun.crypto.provider) from the serialized structure. With compression the cost decreases to 5kB.

The optimized implementation provides the optimal cost. Going into state $P$ costs 5 bytes of header, 160 of symmetric keys and 2 bytes of cipher suite identifier. Going into state $T$ costs

\[\text{readMAC, writeMAC, readCipher, writeCipher, see sun.security.ssl.SSLEngineImpl in } \text{http://grepcode.com}\]
4 TLS SWITCHING IMPLEMENTATION

- **Renegotiation**
- **Initial Handshake**
- **Going into state P**
- **Going into state T**

<table>
<thead>
<tr>
<th></th>
<th>Impl. 1</th>
<th>Impl. 2</th>
<th>Impl. 1</th>
<th>Impl. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sent</td>
<td>479 (100%)</td>
<td>350 (73.07%)</td>
<td>13,436 (2,805%)</td>
<td>167 (34.64%)</td>
</tr>
<tr>
<td>Received</td>
<td>383 (100%)</td>
<td>1428 (372.85%)</td>
<td>0 (0%)</td>
<td>0 (0%)</td>
</tr>
<tr>
<td>Total</td>
<td>862 (100%)</td>
<td>1778 (206.26%)</td>
<td>13,436 (1,559%)</td>
<td>167 (19.37%)</td>
</tr>
</tbody>
</table>

Table 1: Cost in byte of different operations on the proxy-trusted service link.

- 5 bytes of header and from 2 to 16 bytes of sequence numbers by removing non-significant 0 digit.

Contrary to the naive implementation, the additional costs of the optimized implementation are negligible compared to the costs of renegotiations. The naive implementation’s cost (i.e., 16 times that of a renegotiation) could increase the latency of small trusted devices, e.g. by 31ms with a 424kB/s NFC connection.

4.4 Discussion about implementations

The naive implementation requires the proxy and the trusted service to use the same TLS library, thus the proxy and the trusted service are interdependent. On the contrary, the optimized implementation allows the proxy and the trusted service to use different TLS libraries, the proxy and the trusted service are independent.

The optimized implementation modifies classes on the fly to extract symmetric keys when renegotiating. Modifying and rebuilding TLS libraries is not acceptable as it requires to be repeated at each library update. Re-implementing a whole TLS layer, is dangerous as it could lead to several security threats due to implementation errors or slow reactivity. Smart-cards use a subset of Java and require optimized code, thus an own-made TLS stack could be used or modified instead of modifying classes on the fly. Moreover, some TLS libraries, such as JSSE, do not provide access to TLS extensions features.

To avoid performing a TLS renegotiation when unrequired, the proxy could be the one performing the initial handshake, in place of the trusted service. Thus, TLS connections start in state P and will not require TLS Switching as long as no sensitive information are exchanged, or expected to be exchanged. Then, to go into state T for the first time, the proxy sends a p, instead of a t message to the trusted service.

Modified Instead of exchanging ciphers information, the proxy and the trusted service could cooperate to perform a renegotiation. To decrease the amount of data exchanged between the proxy and the trusted service, TLS Handshake proxying techniques may be used to perform, on the proxy, renegotiations when going into state T or the initial handshake. Although TLS Handshake proxying cannot be used for renegotiations when going into state P as the proxy cannot generate TLS records during state T. Even though some bytes might be gained, this requires to modify parts of TLS libraries, and leads to make the trusted service and the proxy interdependent.

Modified Cooperation could also be made by having one generating and reading clear renegotiation messages and the other encrypting and decrypting them with the current keys. Such feature would cost at least one renegotiation to go into state T, and two (asking to encrypt/decrypt a renegotiation message and receiving the result) to go into state P. The additional cost of TLS Switching is then 0% of a renegotiation when going into state T, and 100% when going into state P, against 2.4% and 19.37% for the optimized implementation. We consider
that gaining 7 to 21 bytes when going into state T is not significant.

5 Proxy Architecture

The proxy should not enforce a communication protocol in any of its communication interfaces.

The communication interface towards the client (called client interface) should offer several proxy protocols to the client application such as SOCKS [21] or HTTP Proxy [11]. SOCKS v5 is easily implementable and supports TCP and UDP streams. HTTP Proxy only supports HTTP streams. For client applications non-compliant with proxy protocols, tools can be used to "socksify" [7] them. Static redirections could be offered, for example, streams received on port X redirected to the server Y on port Z. The client interface should allow trivial insertion, substitution and deletion of proxy protocols, making the proxy architecture easily extensible.

The communication interface towards the server (server interface) should be configurable to use a network proxy if the client platform is behind one.

The communication interface towards the trusted service has two-layers. The first layer encapsulates the communication medium, and the second abstracts the format of the message. The first layer allows trivial insertions, deletions and substitutions of communication mediums such as USB, WiFi, Bluetooth or NFC.

The second layer uses a slave/master communication: the master (the proxy) sends a command and the slave (the trusted service) must immediately respond. The slave/master communication is intended to be compliant with slave/master mediums, such as ISO7816-compliant smartcards, like the MasterCard’s Display Card [8].

---

Commands and responses should be human-readable and writable to facilitate debugging operations. In consequence, a human is able to monitor and understand the communication between the proxy and the trusted service with tools such as Wireshark,\(^9\) or to simulate the proxy (or the trusted service) to test the trusted service (or the proxy) with tools such as netcat\(^10\) or openssl s_client/s_server.\(^11\) To optimize the interpretation of commands (and responses), messages start with an ASCII command-name (or response-status) followed by optional arguments. Response-status could use the HTTP status-codes [11].

### Listing 2: Examples of messages between the proxy and the trusted service

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CONNECT <a href="http://www.sigsac.org:443">www.sigsac.org:443</a></td>
</tr>
<tr>
<td>2</td>
<td>200 OK</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>CONNECT <a href="http://www.sigsac.com:443">www.sigsac.com:443</a></td>
</tr>
<tr>
<td>5</td>
<td>403 FORBIDDEN 2-Phishing:Requested domain too close from known domains.</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>MAKE_ME_COFFEE</td>
</tr>
<tr>
<td>8</td>
<td>418 I_AM_A_TEAPOT</td>
</tr>
</tbody>
</table>

The trusted service can choose to either answer immediately to the proxy’s commands or ask the proxy to wait, in which case the proxy performs an active waiting by sending regularly a specific command to the trusted service.

Theses communications should be configurable via a JSON [28] or XML\(^12\) [24] configuration file. JSON and XML files are human-readable and are extendibles as fields can be added and ignored by non-compliant readers. Since theses formats are standardized, APIs and libraries are provided for common programming languages to write, read and edit such files. In Java, the XML API is provided by the javax.xml.parsers package\(^13\).

### 5.2 Filters and commands

Incoming data from client or server are processed by a set of Filters. Thus, by adding or removing a filter, features can be added or removed from the proxy.

If needed during the processing of a messages, a filter can request information or services from the trusted device via a command, such as asking the trusted device to send sensitive information to the server.

The trusted service has a set of Commands. Similarly, by adding or removing a command, features can be added or removed from the implementation of the trusted service. When a command message is received from the proxy, the trusted service invokes the command matching the name of the ASCII command.

The implementation of Commands follow a command design pattern whereas Filters follow the chain of responsibility design-pattern [14, Chap.5].

---

\(^9\)https://www.wireshark.org/
\(^10\)nc(1) – Linux man page, http://linux.die.net/man/1/nc
\(^11\)s_client(1) – Linux man page, http://linux.die.net/man/1/s_client
\(^12\)https://www.w3.org/XML/
\(^13\)https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/package-summary.html
6 Applications of TLS Switching

Modified TLS Switching is developed to allow sending or receiving sensitive information without exposing them on the client-side. As the trusted device performs a TLS certificate verification, TLS Switching guarantees the origin (or destination) of sensitive information during state T. The origin (or destination) is also granted during state P if the proxy is considered secure enough to not leak session keys and IV. In this way phishing attacks by dubious certificate authorities on the client-side are prevented.

To send or receive sensitive information, the TLS connection must be in state T, thus the proxy and the trusted service must implement a mechanism to decide when to change to one state or the other.

Added Please note note that as OTDP is transparent to the server, the server is unable to know the current connection state, and thus to which it sends TLS records or from whom it receives TLS records. To do so, the server must authenticate the trusted services as seen in section 3.3.

6.1 Receiving sensitive information

Predicting when the server needs to send sensitive information is not possible in a generic way. Although message verifications and general computations are possible, the trusted device is not meant to replace the client [19], and therefore should limit what the user is allowed to view or edit.

To send sensitive information, either the server must interact with the trusted service to ask for it, or the trusted service has to predict when sensitive information is being sent.

Asking the trusted service to switch to state T can be done either by asking for client authentication during handshake or with a TLS extension in the Server Hello (if proposed by the Client Hello). Renegotiation can be asked by the server with a Request Client Hello.

Predictions must be performed by the trusted service, i.e. during state T. After establishing a TLS connection or giving sensitive information, the trusted service has to decide whenever the connection have to stay in state T to receive sensitive information, or may go into state P to optimize the server-client communication.

One example of sensitive information is server certificates. Certificates are sent during the TLS connection establishment and must be verified by the trusted service to ensure the TLS connection security.

6.2 Sending sensitive information

Information that is supposed to be sensitive can either be known to the user, to the trusted service, or be generated. Information known by the user (like a password or fingerprint) should only be given to the trusted device, which in turn should know when to request it. This information should not be given to the client or the proxy.

Detecting when sensitive information is being sent is performed by the proxy. Even if corrupted, the proxy is unable to answer the server without the cooperation of the trusted service. Such detection can be performed in three ways: detecting a specific token in a client message, identifying a client message, or identifying a server message.
6.2.1 Detecting a specific token

Instead of giving sensitive information, the user writes a specific token such as $\text{trust:pwd}$ (as shown in Figure 7). The proxy then detects the token and gives the message to the trusted service which makes the TLS connexion going into state T, asks user confirmation, replaces the token by the requested sensitive information and sends the message to the server application. Sensitive information can also be generated by the trusted service, such as passwords during registrations, instead of asking them from the user.

The specific token must be accepted by the client application field format checking. As the token can be transformed by the client application, for example by hashing it, the proxy is only able to detect tokens transformed in a predictable way, using secrets known by the proxy.

Detected tokens could have been provided by the user, or accidentally generated. Therefore tokens should not be widespread tokens and be long enough to reduce the probability of appearing by accident.

Replacing a token by a non-identical length information could lead to errors if the message contains its own length.

If the proxy is corrupted, tokens could be moved, making the trusted service writing the sensitive information at a different place in the message which could lead the server to expose the sensitive information.

Knowing the message format, token seeking and replacing can be improved. For example, in HTTP messages, the message length could be updated in the HTTP header, and tokens be searched, and replaced, in GET and POST fields with verification of the field name.

6.2.2 Identifying client messages

Instead of seeking tokens, the proxy may identify client messages with a specific form. The user can thus just enter random values in the client application, as the proxy will match the message with a known template and sends it to the trusted service to insert the proper sensitive values.

For all protocols, messages used to carry sensitive information must be known and identifiable by the proxy and the trusted service. For such feature, Deep Packet Inspection methods could be employed [1].

As this method does not use tokens, there is no issues related to format checking, transformed or accidental values, value-length changes, or writing of information in wrong places.

Identifying messages before sending them to the server could be used with black and white-lists stored in the trusted device to block unwanted messages. For example to block HTTP requests to phishing websites in order to protect user’s sensitive information or even HTTP requests to advertising servers to protect user’s privacy.
6.2.3 Identifying server messages

Instead of client messages, server messages may be identified, as shown in Figure 8. The proxy detects and intercepts the server information requests, transmits them to the trusted service which in turn responds to the server application. Users are not required to fill in fields with random values or specific tokens.

Users may not want to respond straightaway to the server information request. For example a website can propose services to anonymous users and require users to log in in order to access additional services. Then users are not urged to log in and can chose whenever they want to. Such situation may be detected to propose users to log in (or send sensitive information) whenever they want to without performing actions on the client application, and without preventing them to use other server’s services. Although some client applications could be incompatible with this method, for example if they block until users provide the sensitive informations on a client application’s form.

Such method is already used for HTTP DAA with HTTP XDAA.

6.3 Trusted device and server authentications

TLS handshake protocol offers client and server authentications. The trusted service may be authenticated by the server during handshake, with client authentication. Thus the trusted device must have a private key stored in the trusted device or computed from user information, such as passwords or biometric data. The trusted service may propose other authentication methods through TLS extensions in Client Hello.

The trusted service must ensure the origin and destination of sensitive information by, e.g., authenticating the server. For this, the trusted service must check the server certificate and search the requested IP and/or domain name in the blacklist and the whitelist. In order to prevent phishing attacks, the user can be warned by the trusted service when requesting for a domain-name too close to one from the whitelist.

DNSSEC [4] may also be used by the trusted service to prevent attacks against the DNS protocol [5].
6.4 TLS Switching based protocols

Although TLS Switching is meant to be compliant to existing client and server applications and protocols, protocols based on TLS Switching may be designed. An example can be a server application that could explicitly ask for sensitive information to the trusted device over TLS Switching.

Such protocols should remain transparent to the client application and should not limit services access to users without an OTDP architecture.

7 User and account management

![Figure 9: User Management](image)

7.1 Profiles

The trusted service can provide user-centric identity management [16, 17]. We assume and detail such a scenario in this section and refer to Figure 9 for an illustration. A trusted device could be shared between several users, such as members of a family or a company. Thus an "admin" user is needed to manage the users accounts on the trusted service. A "guest" user, administrated by the admin, may be created for a temporary user.
The trusted service can be used by a particular user in several contexts such as professional, family, or privacy sensitive contexts. The user should have a set of "profiles" she can enable or disable depending on the context, thus protecting the user’s privacy by context isolation. A profile (identified though a profile ID) contains a black-list, a white-list, and a set of identities.

An identity contains all the information needed by the trusted device to authenticate the user to a server application. The required information depend on the authentication protocol and can be, for example, a token, a hash, a login, or a password.

7.2 Backup, export and import

The trusted device contains sensitive information used to access resources on servers. These information must not be lost if the trusted device is lost, stolen, broken or replaced. Otherwise, the user will not be able to access these resources any more.

The trusted device information should be stored elsewhere to allow their recovery on the same or on another trusted device. As these information are sensitive, their confidentiality, integrity and authenticity must be protected when exported, and verified when imported. Export and import must require user confirmation.

7.3 Sharing profiles and company management

Users could need to share profiles with other users, on the same or another trusted device. A user could share her white and black lists or some of her identities with other users. For example a company can have one server account for several employees.

The user who create the profile is the profile owner and can decide to share it with users who accept a sharing request. Then, profile modifications could follow several policies, depending on the needs:

- The profile could be unmodifiable once created, modifiable by the owner or modifiable by all its users.
- Updates may require confirmation from the owner or the users, a changelog could be shown or the update could be transparent.
- The update could be manual, a regularly synchronization, or a synchronization at each changes.

Shared profiles could be copied for each user, sharing and exposing sensitive information on the users trusted device. Otherwise, the sensitive information could be retrieved by querying them from the owner’s trusted device when needed, for example by using TLS Switching.

Such a system could be used in companies to provide an efficient account management. One or many administrators could manage the company accounts by creating and sharing profiles. This could also prevent some forms of social engineering attacks or careless mistakes such as not changing accounts credentials when an employee quits the company.

8 Usability

Users are often the vulnerability in a holistic system. A secure way to authenticate is not always enough, users must be urged to use it and be prevented from revealing their credentials on the client which can be corrupted.
To be used and not bypassed by the user, the system must be intuitive and ergonomic. Users should also be aware of the trusted service and the proxy actions.

### 8.1 User authentication to the trusted device

The trusted service must verify the user’s identity before granting her access to its services and allowing the proxy to interact with the trusted device. This prevents an attacker from using the trusted device if this is stolen or lost.

Still, if the user is authenticated and the trusted device is lost, then the user should be disconnected from the trusted service. This could be done automatically by the trusted device when it detects that it cannot communicate with the proxy. This can be realistic since the client platform on which the proxy resides is a different physical system than the trusted device. It is less probably that the attacker can obtain both devices. Moreover, we could have the user’s identity verified regularly through continuous authentication techniques based on biometrics or for each action.

To avoid loosing the trusted device, the user could be urged to wear the device with her at all times. For example, the trusted device could have a strap or be a phone case. The device could also notify the user when she gets away, for example if a connection is broken, e.g. bluetooth, between the trusted device and a cloth device wore by the user, e.g. watch, necklace.

### 8.2 User interaction

Users must be warned when the trusted service awaits for user inputs or when messages are blocked.

The trusted device could generate a notification when the user attention is requested, e.g., by ringing or vibrating when it is a smart phone case. To prevent users from disabling the trusted service notifications, notifications must be used with parsimony, only when needed.

As the user focus on the client application, the proxy could be used to display explanations or security messages on the client application, or on the client. This improves the ergonomics, but as the client might be corrupted, the display is not granted. For example, the proxy could respond to the client, in place of the server, with an informative message.
For a web browser, as shown in Figure 10, a fake page could be shown to the user, requesting her to check the trusted device. When the user has performed the requested actions on the trusted service, the page could be automatically refreshed. To achieve such refresh, the fake page sends an AJAX query to the proxy which responds once the trusted service has confirmed the execution of the requested actions. When the AJAX response is received, the fake page can refresh itself.

The proxy could modify the server responses to insert fields or informative messages. As shown in Figure 11, the proxy could insert script code in HTML pages to disable password inputs and fill them with a specific token as seen in section 6.2.1. Thus the user is urged to not fill the password input, which she could have done out of habit [30]. Alternatively, the user should be able to disable the script code to use the input field normally if she wants or needs to. Scripts could be inserted to disable format checking for the fields in which the user might write a token.

The proxy could also be used to let the user input information for the trusted service on the client, which is more ergonomic. The provided information must not be confidential, and must be verified and confirmed by the user.

The user could also interact with the proxy or the trusted device in a browser with a specific URL. For example to configure the proxy or to import/export information stored on the trusted device.

### 8.3 User behaviour

When possible, sensitive information should not be exposed on the user-side as the user might be tempted to give the information to an attacker performing a social engineering attack, or simply write the sensitive information on a post-it. Sensitive information, such as passwords, could be generated on the trusted device and never shown to the user. As users do not have to remember the generated information, we can generate safer secrets by having more entropy. Still, the generated information must be compliant with the format required by the server applications, for examples rules can be enforced in a password creation s.a. "no special characters allowed".

Users do not change regularly their passwords as they should do. Stored passwords and other sensitive information could have an expiration date which when reached would warn the user. With the user’s approval, the trusted service could replace outdated information and update them on the server. Still, this is not a generic way of doing this because each server application use their own protocol to update information.

The user should be warned when the trusted service logs her in a server application; the trusted service should at least ask for user confirmation. As several identifies can be suitable for a given authentication request, the trusted service can ask the user to choose the identity to use between the suitable identities and consider this choice as a confirmation.
9 Trusted device used for the OTDP architecture

To implement our OTDP architecture, we use the OffPAD [29], a phone-cover trusted device connected to a smart phone through microUSB. A strong point of the OffPAD is that it is integrated into an object already carried by the user and thus are constantly with the user. So that it is less likely the user left the trusted device, e.g. on her working place. Also, the fact that the user does not have an extra object to carry in her pocket should improve acceptance for such solution.

For our needs, we use the OffPAD’s following features:

- **Secure storage** to store CA certificates and user account data. CA certificates are used in sections 3.3, and 6 to verify servers’ identities. More details about user accounts are given in section 7.
- **Secure screen** used to convey information to the user, such as notifying her, warning her or asking her for confirmation, see section 8.3.
- **Two secure buttons** asking the user for confirmations, see section 8.3.
- **Micro-USB communication** between the proxy and the trusted device, assuming that the USB connection is secured.
- **TLS Switching implementation**, see section 4.
- **Secure random generator**, to generate nonce for XDAA.
- **Fingerprint sensor**, to authenticate the user, see section 8.1.

10 Related Work

10.1 TLS Handshake Proxying

TLS Handshake Proxying [26] is a technique used with edge servers that cache content to reduce the connection latency between a client and a server, i.e., the client establishes a TLS connection to the local edge instead of the distant server. However, the edge is not trusted to share the server’s private key. In consequence, during the local TLS handshake the edge will forward to the server the message involving the private key, i.e., only one long-distance communication.

This TLS Handshake proxying techniques may be used in our architecture, but only when in state P, as our proxy cannot generate TLS records during state T. Nevertheless, this requires to modify parts of TLS libraries, and makes the trusted device and the proxy interdependent.

10.2 TLS Splitting

TLS Splitting [22] uses a proxy to split a non-encrypted TLS connection into two streams, one for the content and one for the content’s integrity. The client establishes a non-encrypted TLS connection with the server through the proxy which forwards all messages. The server answers with the MAC of the content and a content identifier. Then the proxy replaces the identifier with the cached content before sending the TLS record to the client. For encrypted connections [22] proposes a work-around by sending the symmetric encryptions keys and initialization vectors
to the proxy so that it can read TLS records, and write records knowing the MAC. Thus, the proxy is trusted for confidentiality, but not for integrity.

In our case, the proxy needs to be able to build complete TLS records when in state P, but is not trusted to read or write when in state T, i.e., when the trusted channel is used.

11 Futur work

We plan to implement a version of OTDP using TLS Switching to add TLS support to the proxy.

TLS features and extensions support in TLS Switching could be tested and improved, as for TLS resumption widely used in HTTP(S) connections.

OTDP architecture could be tested by implementing support for known authentication protocols and services.

OTDP architecture could also be generalized. The OTDP Proxy must be between the client and the server applications, and should run on the client. However, it may be outsourced in an untrusted go-between, such as company gateways or an user internet box. The trusted service should runs on the trusted device. However, it may be outsourced in a trusted server and the trusted device be only used to perform trusted input/output exchange with the user. For example the trusted server could ask confirmation to the user by sending a SMS on the user’s phone, considered here as a trusted device in a scheme similar to 3D-Secure.

Using cooperation between the Proxy and the Trusted service when going into state T could be implemented in TLS Switching.

12 Discussion and Conclusion

We have presented a new architecture that uses a Proxy component deployed on the client platform (this is assumed to be corruptible) and a Trusted Offline Device that can communicate only with the proxy to provide various security sensitive functionalities. We called this a Offline Trusted Device Proxy (OTDP) architecture to emphasize these two components and their relationship. This architecture combines two more traditional architectures, as explained in Section ??.

We introduced the concept of TLS Switching, which allowed the OTDP architecture to switch between two states of TLS encryption: one state T where a TLS connection was made between the trusted device and the server (for transmitting sensitive information); and a state P where the TLS connection was used by the Proxy to transmit the communication between the client and the server, as normally done.

The OTDP architecture is designed to be flexible to different protocols and application scenarios. The implementation is designed to allow easy modification or addition of new features through the use of the Filters and Commands design patterns, which fit very well in this situation.

We have presented several possible applications of this architecture in Sections 6 and 7. Other applications could be imagined. With OTDP we could parse e-mails to prevent email phishing which ask the victims to responds with their credentials. We could sign documents like e-mail enclosures or medical receipts. We could grant access to document to other users using
cryptographic techniques. We could also use the trusted service to authenticate to a client application instead of authenticating to a server application.

Since we assume the client platform to be corrupted, the presence of a proxy on the client-side does not lower the security. One could think of an increased attack surface as the attacker may exploit JVM or proxy implementation vulnerabilities as well. Moreover, we do not have any guaranty that the proxy will not be bypassed. Nevertheless, this problem cannot be avoided even if the trusted device sits between the client and the server, as corrupted client platforms can bypass it in the same way. A user cannot know for sure whether the trusted device is between the client and the server, he can only believe that. By not offering this guaranty, our OTDP architecture raise awareness on the user’s side, which is better than a false feeling of security.

We assume that the trusted device is secure, but it might have hardware or software backdoors or some spyware put by the manufacturer or the distributor. As we use a proxy, a user can install her own version (or inspect our code) and be sure about what the Proxy does. Moreover, the proxy may choose to not send all data to the trusted service, thus reducing the spying potentiality. The proxy can also be instrumented to monitor (to some extent) the actions of the trusted device, as this is assumed to be offline (only communication with the proxy). All these assume that the Proxy is in the control of the user and not of an attacker.

We also cannot prevent an authorized user to give access to an unauthorized user. In a scenario where the credentials exist only on the trusted device, the use can only give away this device to the unauthorized user, instead of give credentials through e-mail or phone. Moreover, when the trusted device is designed to be attached to the user, e.g., as the back cover of the phone, which may also do continuous authentication, sharing the trusted device becomes even less possible.

References

REFERENCES


