entropy

MDPI

Article

A Framework to Secure the Development and Auditing of SSL Pinning in Mobile Applications: The Case of Android Devices

Francisco José Ramírez-López [1], Ángel Jesús Varela-Vaca [2], Jorge Ropero [1], Joaquín Luque [1] and Alejandro Carrasco [1,*]

1 Departamento de Tecnología Electrónica, Universidad de Sevilla, 41012 Sevilla, Spain; framirez4@us.es (F.J.R.-L.); jropero@us.es (J.R.); jluque@us.es (J.L.)
2 Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla, 41012 Sevilla, Spain; ajvarela@us.es
* Correspondence: acarrasco@us.es

Received: 23 September 2019; Accepted: 20 November 2019; Published: 21 November 2019

check for updates

Abstract: The use of mobile devices has undergone rapid growth in recent years. However, on some occasions, security has been neglected when developing applications. SSL/TLS has been used for years to secure communications although it is not a vulnerability-free protocol. One of the most common vulnerabilities is SSL pinning bypassing. This paper first describes some security controls to help protect against SSL pinning bypassing. Subsequently, some existing methods for bypassing are presented and two new methods are defined. We performed some experiments to check the use of security controls in widely used applications, and applied SSL pinning bypassing methods. Finally, we created an applicability framework, relating the implemented security controls and the methods that are applicable. This framework provides a guideline for pentesters and app developers.

Keywords: SSL pinning; security; mobile applications; Android; auditing; vulnerabilities; OWASP

1. Introduction

Nowadays, the use of mobile devices is constantly increasing to do the same operations that used to be done using web services less than a decade ago [1,2]. However, it is necessary to provide the same security solutions in both environments since both operations are equally critical. SSL/TLS (Secure Socket Layer/Transport Layer Security) technology has been widely used as the basis to secure many aspects of the Internet, such as securing HTTP protocol [3].

In 2018, 1.54 billion smartphones were sold [4]. Thus, the rapid growth in the number of smartphones has led to an increase in security threats related to them [5]. Day by day, we read cases of users who have been scammed through the use of mobile applications [6]. For example, users may download some modified version of an application (hereinafter, app) that is not controlled by the owner. In some cases, access to sensitive information by other users of the app has also been detected. This is because many of the controls that have been applied in the web environment have not been considered in the mobile environment. Moreover, several mobile apps do not even implement SSL/TLS validations [7]. Even when using SSL/TLS, apps may have vulnerabilities, especially to Man-in-the-middle (MiTM) attacks [8]. Other attacks, such as replay attack, eavesdropping, and session hijacking, are also common [3]. Security controls such as SSL pinning are desirable [9] but always optional. SSL pinning, also known as certificate pinning or SSL/TLS validations, allows clients to have greater confidence that the certificate used by a server is not compromised [10]. Nevertheless, it is also possible to circumvent SSL/TLS validations [7]. Thus, SSL pinning can be skipped.

Trying to consider the particularities of mobile apps, the OWASP (Open Web Application Security Project) Mobile Application Security Verification Standard (MASVS) is an attempt to standardise these requirements using verification levels that fit different threat scenarios [11]. One of the most important challenges in mobile app security is to protect data flows over insecure communication channels [12]. Insecure communications include poor handshaking, incorrect SSL versions, weak negotiation or clear text communication of Personally Identifiable Information (PII) [13]. In this paper, we explain how an app can be fortified considering some security controls, shielding some aspects to avoid attacks.

This paper first analyses the vulnerabilities of SSL/TLS implementations and the necessity of SSL pinning techniques. Then, we propose a set of security controls that can prevent an app from suffering SSL pinning bypassing. Afterwards, we selected a set of popular Android apps to check if the latter bypassing methods, described in [7] are still valid. We also propose two new bypassing methods, and we have applied them to the app set. Finally, we have defined a framework for the applicability of bypassing methods depending on the security controls that have been applied by the app developers.

The rest of the paper is structured as follows. Section 2 deals with the background of this paper. Concepts such as the OWASP Mobile Testing Guide, the SSL/TLS protocol, the SSL pinning technique and the problem of bypassing of SSL pinning are introduced. Section 3 presents a set of security controls to protect apps against bypassing methods. In Section 4, we evaluate our approach by providing experiments. The obtained results are analysed in Section 5. Finally, the paper is summarised, and the conclusions and future work are presented in Section 6.

2. Background

This section presents some fundamental concepts to set the context of this paper. First, the OWASP Mobile Testing Guide for mobile applications is introduced, justifying its use as a security model for web applications. Secondly, the operation of SSL/TLS is briefly described. SSL/TLS has been used for years to add an extra layer of security. Finally, some SSL/TLS vulnerabilities are introduced, so that we justify the need of this research.

2.1. OWASP Mobile

OWASP is the worldwide organisation responsible for generating a standard for security in web apps [14]. This way, we can find several sources of information and methodologies in the OWASP documentation. The best-known methodology is the so-called Top 10, where the most frequent vulnerabilities are shown. OWASP group develop Top 10 security risks for web, mobile and IoT software [15]. Based on our experience, we choose OWASP Top 10 Mobile as our starting point. Table 1 shows OWASP Mobile Top 10 in December 2016, which is the last update. [13].

Table 1. OWASP Top 10.

Category	Name
M1	Improper Platform Usage
M2	Insecure Data Storage
M3	Insecure Communication
M4	Insecure Authentication
M5	Insufficient Cryptography
M6	Insecure Authorisation
M7	Client Code Quality
M8	Code Tampering
M9	Reverse Engineering
M10	Extraneous Functionality

As shown, improper platform usage is considered the most relevant security risk. This category covers the security control that is part of the mobile operating system. However, insecure communication ranks #3 in OWASP Top 10, thus it is also quite an important topic to be considered. SSL pinning is included in this category.

Although there are some other methodologies or lists of controls where apps may be reviewed, we focus on the Mobile Application Security Verification Standard (MASVS), which is defined in the OWASP Testing Guide. The OWASP Mobile Testing Guide was recently published in its first version [11]. In this guide, security controls are defined and can be reviewed according to different categories. Every control is described. It is also shown how it can be tested. Finally, the guide sometimes offers a solution to the problem described by the control. However, the solution must be adapted to the system or the client that is audited.

Within the OWASP controls, several control layers must be considered. The utilisation of these layers depends on the app. There are three existing layers, called verification levels: L1, standard security; L2, defense-in-depth; and R, resiliency against reverse engineering and tampering.

All the controls are grouped into categories. Within each category, different security controls must be applied. We focus on category V5, network communication requirements. To secure network communication, we should follow the recommendations shown in Table 2.

Table 2. Network communication security verification requirements.

Control	Description
5.1	Data are encrypted on the network using TLS. The secure channel is used consistently throughout the app
5.2	The TLS settings are in line with current best practices, or as close as possible if the mobile operating system does not support the recommended standards
5.3	The app verifies the X.509 certificate of the remote endpoint when the secure channel is established. Only certificates signed by a trusted CA are accepted
5.4	The app uses its own certificate store, or pins the endpoint certificate or public key, and subsequently does not establish the connection with endpoints that offer a different certificate or key, even if signed by a trusted CA
5.5	The app does not rely on a single insecure communication channel (email or SMS) for critical operations, such as enrollments and account recovery

To achieve level L1 requirements, Controls 5.1–5.3 must be secured, while Controls 5.4 and 5.5 refer to level L2. However, accomplishing the requirements of Control 5.4 implies accomplishing the three previous controls. Thus, securing Control 5.4 implies achieving L1 requirements. In practice, we can identify this control with SSL pinning.

2.2. SSL and SSL Pinning

Secure Socket Layer (SSL) [16] protocol and Transport Layer Security (TLS) [17] protocol are widely used to provide confidentiality, authentication, and integrity in data communications. SSL/TLS provides three main security services: confidentiality, by encrypting data; message integrity, by using a message authentication code (MAC); and authentication, through digital signatures.

SSL/TLS allows the authentication of both parties, server authentication with an unauthenticated client, and total anonymity. The authentication of client and server may be carried out through digital signatures. Nowadays, digital signatures are mostly based on certificates (i.e., X.509 standard) or shared keys. In the case of using certificates, they always have to be verified to ensure proper signing by a trusted Certificate Authority (CA). On the other hand, these protocols also provide anonymous authentication by using Diffie–Hellman for key exchange from SSLv3.0, TLSv1.0 and later versions [18].

SSL/TSL protocol is based on a handshake sequence whose main features [19] are used by client and server, as follows: (1) Negotiate the cipher suite to be used during data transfer, and exchange random numbers (master key); (2) Establish and share a session ID between client and server; (3) Authenticate the server to the client; and (4) Authenticate the client to the server.

Several providers are widely used such as JSSE (Java Security Socket Extension) [20], OpenSSL [21], LibreSSL [22], or GnuTLS [23]. There is even specific hardware with built-in SSL/TLS solutions, such as in iOS devices [24].

As mentioned above, one of the Top-3 risks identified by OWASP is insecure communication due to poor configuration of an SSL/TLS channel. However, SSL/TLS is not a vulnerability-free protocol [25]. SSL/TLS attacks may come from MiTM Attacks, against the handshake or from the SSL implementation. Moreover, it is demonstrated that most of the Android applications use the default SSL implementations provided by Android [25,26]. Similarly, most of the applications are susceptible to be attacked by intercepting communications by MiTM attacks. Thus, SSL/TLS implementations can be broken by MiTM attacks [15]. When the client is in a public network, it is possible for an attacker to be in the middle of the communication and impersonate the server.MiTM attacks occur due to the lack of validation or an incorrect validation in the protocol.

In SSL/TLS, certificates are verified to check whether they are signed by proper CA. With a MiTM attack using a spoofed certificate, SSL/TLS may be mislead (see Figure 1). If this is done with a valid certificate, the client's system checks the certificate and considers it valid. Then, the attacker may capture all the plaintext data that are exchanged between client and server. The certificate is trusted, even though its origin is unknown. Once the certificates are accepted and the handshake is finished, the SSL/TLS communication is established as secure. Meanwhile, a third party is bypassing the channel intercepting and deciphering all the packets in the communication.

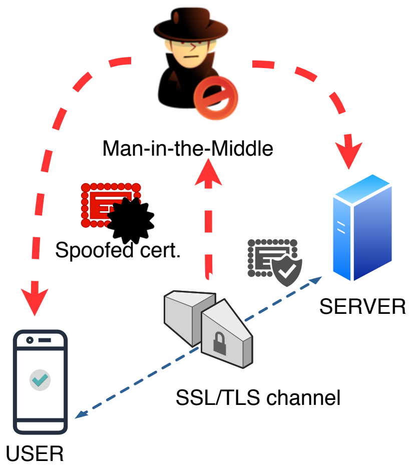


Figure 1. Bypassing SSL/TLS by using spoofed certificates.

The pinning technique or HTTP Public Key Pinning (HPKP) [7] has emerged in the last years as a security control to fortify HTTPS-based pinnings against MiTM attacks. The SSL pinning techniques have been widely used as a complement to enforce the security of SSL/TLS communications for mobile applications [9].

Figure 2 shows the SSL pinning implementation process, which is divided into two stages. In the first stage, the mobile device must initiate communication with the server. The server responds whether it is active or not (server hello in Figure 2). Then, the client asks for the server's certification when the server answers with the content of the information of its certificate and public key (verification certificates in Figure 2).

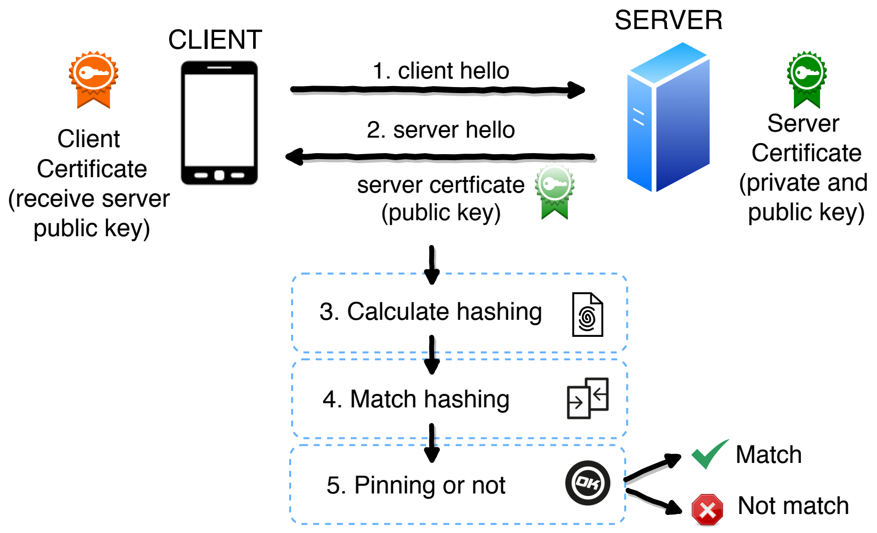

CLIENT
SERVER
Client Certificate (receive server public key)
1. client hello
2. server hello
server certficate (public key)
Server Certificate (private and public key)
3. Calculate hashing
4. Match hashing
5. Pinning or not
Match
Not match

Figure 2. Certificate verification process and pinning.

The second stage is called pinning. The mobile device follows a verification process where the certificate is received from the server. When the client receives a message from the server, it checks its authenticity using the server's public key, which is stored in the client. The received public key has to match the one that is stored. If so, the client opens a negotiation or sends packages signed with that public key. When the client does not coincide, it cuts off the communication. Thus, it does not send anything to the server.

SSL pinning techniques has been proved as good countermeasures against MiTM attacks [27–29]. Most of the works in the literature focused on improving SSL/TLS implementations. The utilization of SSL pinning techniques against weak and default implementations of secure communications and MiTM attacks is usual. However, SSL pinning techniques are not invulnerable since it can be circumvented.

2.3. Bypassing SSL Pinning

SSL pinning is also vulnerable when it is not well implemented. Although bypassing techniques are quite well-known in the professional world, there are not many studies about bypassing SSL pinning. Andzakovic used reverse engineering to bypass SSL pinning on Android, but the technique was applied to an only app [30]. Sierra and Ramirez [31] also used different techniques to bypass certificate pinning, but only achieved it with dynamic library manipulation. D'Orazio et Choo defined five methods for SSL circumventing on iOS, and their study is the starting point for ours [7]. Finally, Anthi and Theodorakopoulos used some techniques to bypass SSL validations in iOS and Android, but could only bypass SSL in iOS apps [5]. Moreover, several methods can be used to bypassing SSL pinning:

- *Dynamic analysis of code.* The code that is being executed in memory is extracted and its behaviour is low-level analysed, evaluating registers, the data that are loaded in memory and functions, among others. An example of a dynamic analysis tool is SSL Unpinning by Xposed framework. This tool takes advantage of the fact that the code that implements SSL Pinning is known. In general, pinning developers use a well-known or common template. In this case, an attacker may guess this and use SSL Unpinning to bypass SSL Pinning in the pinning stage. In addition, tools such as Frida or Cycript enable the modification of some functions of pinning at runtime.
- *Static analysis of code.* It consists in extracting and analysing the app code when not at runtime. As we explain in Section 3, if pinning does not implement anti-tampering or exceptions for the modification of pinning, SSL Pinning functions can be replaced to bypass the pinning process.

3. Security Controls to Protect against Bypassing SSL Pinning

Certain countermeasures, such as security controls, may help an app to avoid the bypassing of the SSL pinning process. With four security measures or controls, apps can be fortified against SSL bypass. Here, we present some good/best practices or guidelines as a set of several steps to implement an adequate security solution to secure SSL Pinning. This way, bypassing SSL pinning is avoided. Although OWASP proposes to use a set of controls to ensure channels of communications, our guideline demonstrates and ensures that, with only three steps, mobile pinnings can be fortified against bypassing SSL pinning, and no more controls need to be checked.

The proposed process (In BPMN, the +−symbol indicates parallel execution of tasks) is shown in Figure 3, which presents the measures that have to be taken to protect an app from SSL pinning bypassing. In the implementation phase of the app, root detection, debug detection and anti-tampering measures must be implemented. Finally, the code must also be obfuscated. All these security measures should be applied to guarantee that SSL pinning is not bypassed. They are described in detail below.

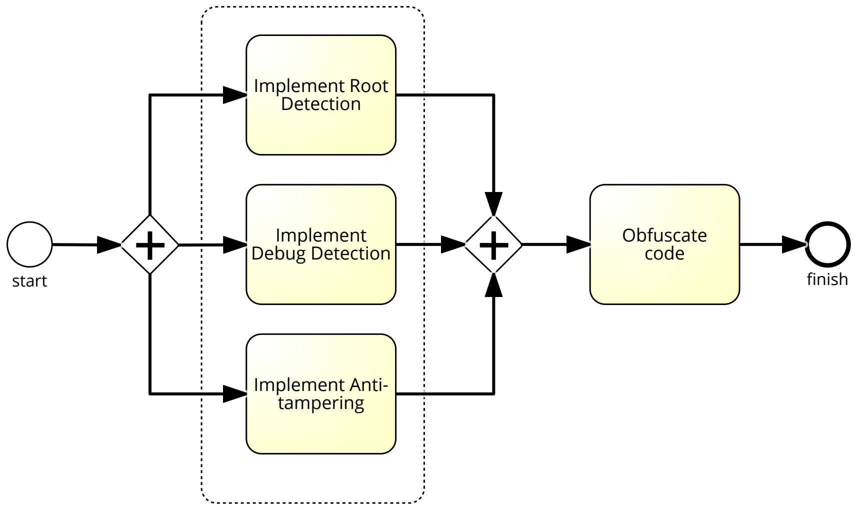


Figure 3. Process to ensure SSL Pinning.

1. **Root Detection** is a security control consisting of checking the blocks of the execution code that turn the pinning into the supervisor mode. Thus, root detection helps prevent the execution of code in supervisor mode. This control improves the effectiveness of reverse engineering and anti-tampering processes.

   There are several ways to detect root [32] but the most common one is checking some files (i.e., binaries and apk files) that are present on a rooted device, such as */system/app/Superuser.apk* or */system/bin/su*. A piece of the code used to check the root execution on the pinning is given in Code 3.1.

   Code 3.1: Example of code to detect root

```
public static boolean checkRoot(){
    for(String pathDir : System.getenv("PATH").split(":")){
        if(new File(pathDir, "su").exists()) {
            return true;
        }
    }
    return false;
}

public static boolean checkRunningProcesses() {
    boolean returnValue = false;
    // Get currently running pinning processes
```

```
        List<RunningServiceInfo> list = manager.getRunningServices(300);
        if(list != null){
            String tempName;
            for(int i=0;i<list.size();++i){
                tempName = list.get(i).process;
                if(tempName.contains("supersu") || tempName.contains("superuser")){
                    returnValue = true;
                }
            }
         }
        return returnValue;
    }
```

2. **Debug Detection** is used to prevent the app code from being controlled. If a device in debug mode is detected, the execution of the pinning is stopped. This measure stops an attacker from seeing the step-by-step behaviour of our code. Together with the obfuscation of code, described below, it allows hiding the internal functioning of the pinning. Functions such as the one we provide in Code 3.2 are used for Android applications. This piece of code is an adaptation of the guidelines OWASP for debug detection, although there are other ways to detect the debug mode.

   Indeed, it is always possible to compile an app in debug mode (i.e., changing the Android Manifest file and modifying the corresponding label). In this case, the entire device is not in debug mode, but the app. Code 3.2 detects debugging mode with different functions: finding the Debugger flag, detecting debugger, detecting if execution has stopped because of debugging and checking running processes.

Code 3.2: Example of code to avoid debugging mode.

```
 public static boolean isDebuggable(Context context){

    return ((context.getApplicationContext().getApplicationInfo().flags &
    ApplicationInfo.FLAG_DEBUGGABLE) != 0);
}

public static boolean detectDebugger() {
        return Debug.isDebuggerConnected();
}

static boolean detect_threadCpuTimeNanos(){
    long start = Debug.threadCpuTimeNanos();
    for(int i=0; i<1000000; ++i)
        continue;
        long stop = Debug.threadCpuTimeNanos();
        if(stop - start < 10000000) {
           return false;
        }
        else {
            return true;
    }
}

public boolean checkRunningProcesses() {
    boolean returnValue = false;

    try {
        Process process = Runtime.getRuntime().exec("ps");
        BufferedReader reader = new BufferedReader(new
        InputStreamReader(process.getInputStream()));
        int read;
        char[] buffer = new char[4096];
        StringBuffer output = new StringBuffer();
        while ((read = reader.read(buffer)) > 0) {
            output.append(buffer, 0, read);
        }
        reader.close();

        // Waits for the command to finish.
        process.waitFor();
        Log.d("fridaserver", output.toString());

        if(output.toString().contains("frida-server")) {
            Log.d("fridaserver","Frida Server process found!" );
            returnValue = true;
        }
    } catch (IOException e) {
    } catch (InterruptedException e) {
    }
    return returnValue;
}
```

3. **Anti-tampering** solution is an important option since any attacker may decompile an application and recompile it. Modifying some parts of the application [32], as previously explained in the SSL pinning bypass process, an attacker could skip the SSL pinning, and thus make the application invalid.

   Code 3.3 gives an example of the code that can be included in an Android application, particularly in the *onCreate* function within *MainActivity*. In this case, the signature of the application is checked only when the app is started. In iOS, the mechanism is similar, as indicated in the Apple Security Transforms Programming Guide [33].

Code 3.3: Example of code to check signatures.

```
private static final int VALID = 0;

private static final int INVALID = 1;

public static int checkAppSignature(Context context) {
    try {
      PackageInfo packageInfo = context.getPackageManager()
          .getPackageInfo(context.getPackageName(),
              PackageManager.GET\_SIGNATURES);
      for (Signature signature : packageInfo.signatures) {
        byte[] signatureBytes = signature.toByteArray();
        MessageDigest md = MessageDigest.getInstance("SHA");
        md.update(signature.toByteArray());
        final String currentSignature = Base64.encodeToString(md.digest(), Base64.DEFAULT);

        Log.d("REMOVE\_ME", "Include this string as a value for SIGNATURE:"
                    + currentSignature);
        //compare signatures
        if (SIGNATURE.equals(currentSignature)){
          return VALID;
        }
      }
    } catch (Exception e) {
    // assumes an issue in checking signature.,
    // but we let the caller decide on what to do.
    }
    return INVALID;
  }

 private void crcTest() throws IOException {
    boolean modified = false;
    // required dex crc value stored as a text string.
    // it could be any invisible layout element
    long dexCrc = Long.parseLong(Main.MyContext.getString(R.string.dex_crc));

    ZipFile zf = new ZipFile(Main.MyContext.getPackageCodePath());
    ZipEntry ze = zf.getEntry("classes.dex");

    if ( ze.getCrc() != dexCrc ) {
        // dex has been modified
        modified = true;
    }
    else {
        // dex not tampered with
        modified = false;
        }
    }

    ...
```

4. **Obfuscate code**. All the measures above do not make sense without preventing any attacker from knowing the code of the app. This way, the analysis of SSL Pinning functions or any of the previous ones that we have presented is more difficult. For this reason, code obfuscation is necessary. There are code obfuscators that convert the existing code into a more illegible one, as seen in Figure 4. Therefore, it is more difficult to detect which are the critical functions of the code for an attacker.

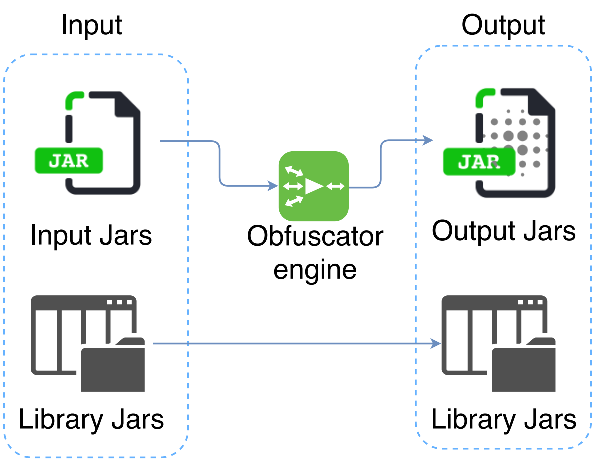


Figure 4. Obfuscation process.

As may be seen, replacing variables and function names for letters and numbers makes it more difficult to guess what a function does. An example of obfuscation is shown in Code 3.4, where names of variables and functions are hidden.

Code 3.4: Example of obfuscation code.

```
public void a(B c){
        switch (C.a()){
        case R.a.b:
        B d = (B) c.a();
        A f = (A) d.c(R.a.j);
        A g = (A) d.c(R.a.k);
        D h = (D) d.c(R.a.m);
        String i = d.b.c ();
        String j = L.a(f.b.c());
        if(!i.equals(String.a)){
                E.c(0);
        } else if (secret.equals(String.f)) {
                d.setTextColor(c.getResources().
                        getColor(R.color.color_nebula));
                f.setTextColor(c.getResources().
                        getColor(R.color.color_nebula));
                ((Vibrator) parent.getContext().
                        getSystemService("vibrator")).
                        vibrate(400);
        D.makeText(c.getContext(),String.m,1).a();
        D.makeText(c.getContext(),String.n,1).a();
        } else {
    }
```

Numerous tools on Android and iOS allow the obfuscation of code, such as *Proguard* [34] for Android or *iXGuard* [35] for iOS.

As aforementioned, these security controls are an easy way to protect the app against bypassing methods of SSL pinning compared to OWASP guidelines. In spite of security controls, several methods can be used to circumvent the SSL/TLS validations.

4. Experiment Setup

Once we have defined some countermeasures to fortify SSL pinning, we must study the methods that can bypass these countermeasures. First, we define a threat scenario. Secondly, we study the current bypassing SSL pinning techniques. Next, we propose new methods of SSL bypassing. Finally, we evaluate our proposal, and provide some guidelines for the correct development and auditing of the applications that use SSL pinning mechanisms.

4.1. Threat Scenario

It is important to remark that neither the app client nor the server is attacked. Communications are the objective. The attacker could install the app in his/her device, decompile the app, modify the app, compile it again and sign it with his/her own certificate. The server does not verify the new certificate, but the app. The attacker could intercept communications, see how communications are implemented and tamper the app. Thus, the integrity of the information is compromised. Figure 5 represents the threat scenario.

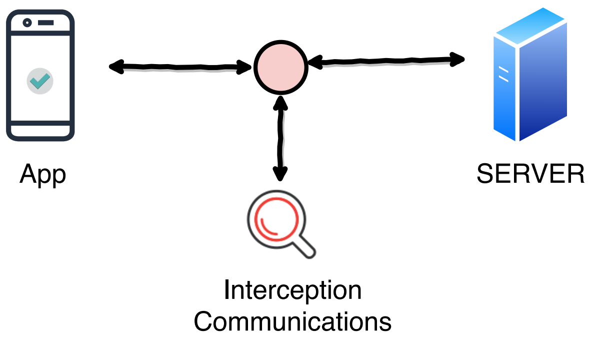


Figure 5. Threat scenario.

For example, if the client uses the app to make a payment, he/she could bypass SSL pinning, see how communications are implemented (if payment is not ciphered) using a MitM attack and modify the values (paying 10 instead of 100 , for instance). Obviously, other features of the app could also be modified.

4.2. Current Bypassing Methods

A MiTM attack can break SSL/TLS pinning, as explained in Section 2.3. Tools such as Burp Suite or Charles Proxy may be used for these attacks. D'Orazio and Choo [7] defined five methods to bypass built-in SSL/TLS validations for iOS devices. They attempted to bypass SSL/TLS validations in 40 popular iOS apps but, surprisingly, only 10 of them performed SSL/TLS validations. The five methods that are used are described below:

1. **Method 1:** Change common name (CM) in certificate. It is similar to the MiTM attack described in Section 2.3, with the proxy changing the CA-signed certificate dynamically.
2. **Method 2:** Insert CA in the application. This method modifies the CA signature that is stored inside the application.
3. **Method 3:** Use SSL Unpinning. Root access is necessary in this case. Tools such as Xposed in Android or SSL Kill Switch are used for SSL bypassing. They take advantage of the fact that the code that implements SSL Pinning is known. App developers use a well-known template, so that an attacker may guess it and use these tools to bypass SSL Pinning in the app.

4. **Method 4:** Debugging app. If the app does not implement anti-tampering or exceptions for the modification of the app, SSL Pinning functions can be replaced to bypass the pinning process.
5. **Method 5:** Modify app executable to avoid the SSL pinning process. The modified version of the app is copied to the device to permanently bypass SSL/TLS pinning.

D'Orazio and Choo's results are shown in Table 3. A checkmark means that the SSL/TLS implementation was bypassed using that method. Otherwise, a dash symbol is used. However, we must highlight two aspects of the results: (1) the authors avoided checking the security controls presented in Section 3; and (2) the tests were only carried out in iOS apps.

Table 3. Summary of SSL/TLS apps bypassed by D'Orazio and Choo [7].

App	**Principal Method**				
	Method 1	**Method 2**	**Method 3**	**Method 4**	**Method 5**
Twitter	-	-	-	✓	✓
NAB Bank app	-	-	-	✓	✓
Dropbox	-	-	✓	✓	✓
WhatsApp	-	-	✓	✓	✓
Bank of America	-	-	-	✓	✓
Facebook	✓	✓	-	✓	✓
ANZ GoMoney	-	-	-	✓	✓
PayPal	✓	-	-	✓	✓
MEGA	-	-	-	✓	✓
CopyApp	✓	✓	-	✓	✓

4.3. New SSL Pinning Bypassing Methods

As a new contribution, we propose two new methods to bypass SSL/TLS validations in Android devices. The first method deals with rooted devices and the second one deals with non-rooted devices. Since the new methods were compared with the ones defined in Section 4.2, we decided to number the new methods as Methods 6 and 7, respectively. Methods 6 and 7 makes circumventing SSL pinning in debug mode possible.

4.3.1. Method 6: Dynamic Debugging of the App in Rooted Devices

To perform this method, it is necessary to use a rooted mobile phone and a remote host. The process is described below:

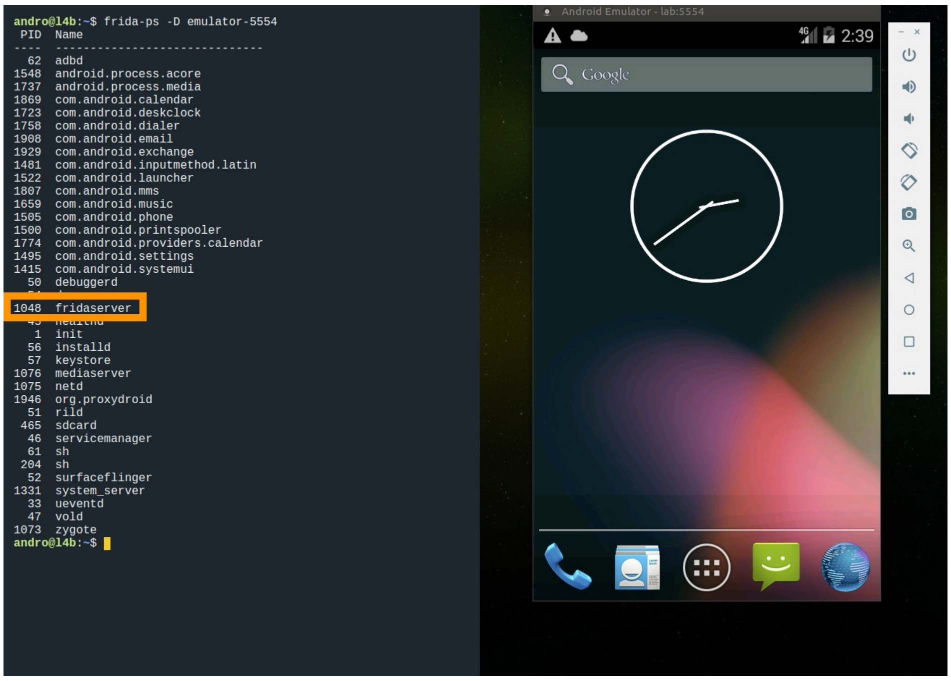
1. The mobile phone must execute a server that acts as a dynamic code interpreter. The server acts as a daemon and it enables interacting dynamically with the physical memory of the device to manipulate and change the behaviour of the executing apps. Thus, the aim of the server is to manipulate or change the expected behaviour of the SSL pinning process. In our case, we used Frida server with the following command:

$$adb \quad shell \quad ./serverscript.sh \tag{1}$$

2. It should be confirmed that the daemon server is running and has enough permissions to read and manipulate the memory of the device. With this aim, Frida framework can be used by means of the following command:

$$frida - ps - U \tag{2}$$

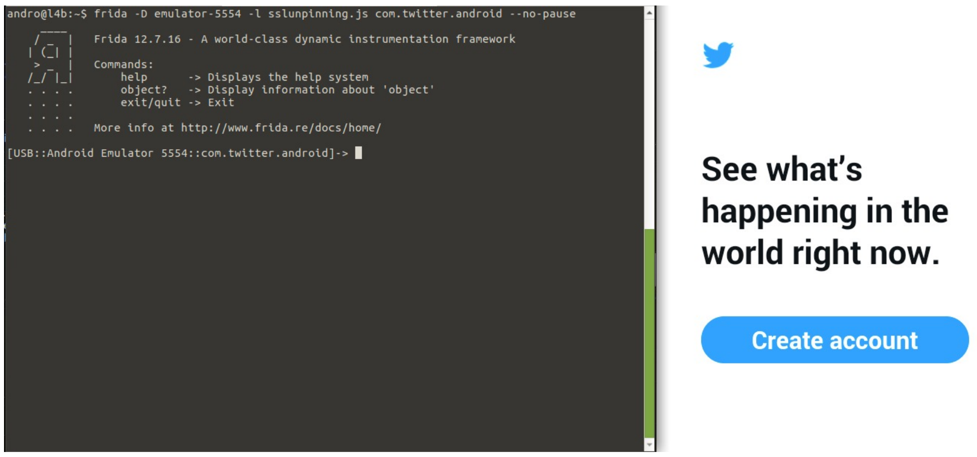
Figure 6 shows how a Frida server is running in a mobile emulator.



```
andro@l4b:~$ frida-ps -D emulator-5554
 PID  Name
----  -----------------------------
  62  adbd
1548  android.process.acore
1737  android.process.media
1869  com.android.calendar
1723  com.android.deskclock
1758  com.android.dialer
1908  com.android.email
1929  com.android.exchange
1481  com.android.inputmethod.latin
1522  com.android.launcher
1807  com.android.mms
1659  com.android.music
1505  com.android.phone
1500  com.android.printspooler
1774  com.android.providers.calendar
1495  com.android.settings
1415  com.android.systemui
  50  debuggerd
1048  fridaserver
   1  init
  56  installd
  57  keystore
1076  mediaserver
1075  netd
1946  org.proxydroid
  51  rild
 465  sdcard
  46  servicemanager
  61  sh
 204  sh
  52  surfaceflinger
1331  system_server
  33  ueventd
  47  vold
1073  zygote
andro@l4b:~$
```

Figure 6. Example of Frida server running in a mobile emulator.

3. Once the daemon server is running on the device, this fact enables the access to the physical memory of the device. This way, we are able to make the appropriate changes in memory to circumvent SSL Pinning measures. It is necessary to indicate which part of the app code must be dynamically modified in memory. To do that, a script must run on the server. In the following, we describe the steps of a script which modifies dynamically in the memory the code. Thus, the target app must be executed on the mobile together with the script to modify its code dynamically. There are several scripts (https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/) that may be used. In our case, the script loads our certificate, creates a KeyStore containing our certificate and generates a TrustManager that trusts the certificate inside our KeyStore. When the app initialises its SSLContext, the parameter TrustManager in the *SSLContext.init()* method is swapped with the one we had created before.
4. It must be checked that there are not any restrictions that prevent the SSL handshake of the connection. If so, the connection is established. Finally, it must be confirmed that the target app works properly and SSL pinning was disabled. Figure 7 shows an example of how the execution of the Twitter app is intercepted by Frida.



```
andro@l4b:~$ frida -D emulator-5554 -l sslunpinning.js com.twitter.android --no-pause
     ____
    / _  |   Frida 12.7.16 - A world-class dynamic instrumentation framework
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at http://www.frida.re/docs/home/

[USB::Android Emulator 5554::com.twitter.android]->
```

Figure 7. Frida intercepting the twitter app execution.

If the app is implementing Root Detection, it is not possible to use Method 6, and Method 7 is used instead.

4.3.2. Method 7: Dynamic Debugging of the App in Non-Rooted Devices

To perform this method, it is not necessary to use a rooted mobile. Steps 1–3 are performed considering that there is no point in rooting the device, as root detection countermeasure is implemented. Hence, it is necessary to skip this countermeasure to disable SSL Pinning. With this aim, the server daemon used in Method 6 must be embedded inside the application. The process is described below:

1. The target app code must be decompiled. For instance, *apktool* may be used.
2. A daemon must be embedded inside the target app. In our case, the daemon is Frida gadget. The function of the daemon is similar to Step 1 in Method 6.
   a. To put in the Frida gadget, an internet permission must be added to the Android manifest. Thereafter, Frida is able to open a socket.
   b. The file containing Frida libraries must be included to the app's library folder.
   c. The *onCreate* method in the main activity of the target app must be modified to force the daemon to be run when the target app starts.
3. The modified app must be re-installed, launched, and its functioning must be checked.
4. From here, Steps 2–4 of Method 6 are executed.

In summary, Figure 8 shows a diagram describing the functioning of Methods 6 and 7.

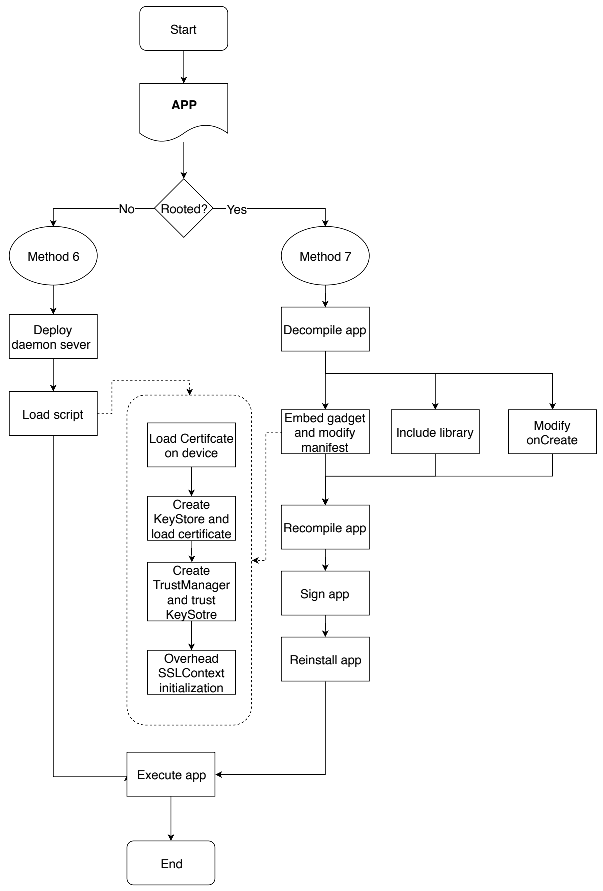

Start
APP
Rooted?
No
Yes
Method 6
Method 7
Deploy daemon sever
Load script
Load Certifcate on device
Create KeyStore and load certificate
Create TrustManager and trust KeySotre
Overhead SSLContext initialization
Decompile app
Embed gadget and modify manifest
Include library
Modify onCreate
Recompile app
Sign app
Reinstall app
Execute app
End

Figure 8. Functioning of SSL Pinning bypassing Methods 6 and 7.

4.4. Experiment Design

To evaluate our proposal, an experiment was designed. Our experiment is an adaptation of D'Orazio and Choo's study [7] presented in Table 3. However, their study neither includes the analysis of security controls nor uses the Android version of the apps. The experiment consists of two phases: (1) checking whether the app includes SSL pinning and contains any security controls (see Section 3); and (2) checking the seven methods presented in the two previous sections. The first phase of our experiment is another novel contribution, while the second phase introduces two new bypassing methods.

To carry out the the first phase of the experiment and to check Methods 1–5, apktool [36], d2j-apk-sign [37], logcat [38] and SSL Unpinning [39] were used. These tools enable changing the certificates and debug the apps.

To perform the new proposal methods, we used Frida toolkit [40]. Frida is a framework designed for developers, security researchers and people who work in reverse engineer apps. It enables the hooking of processes to modify the code in a dynamic way. This allows injecting scripts written in JavaScript to perform certain operations in the app. We used Frida as a server to interact with the apps. To explore app executions, Objection was used. Objection [41] is a toolkit that enables the exploration of mobile devices at runtime. Objection toolkit was used to explore the execution of the apps.

The set of apps we used are the Android version of the apps used in D'Orazio and Choo's study. These apps are ten widely used applications in different fields. Nevertheless, two of the apps (i.e., ANZ GoMoney and CopyApp) were substituted by two similar or more updated ones, Outlook and Instagram. We decided to change those two apps as they are not available in an Android version or their use in our region has been restricted.

5. Analysis of Results

In the next subsection, we present and analyse the results obtained for the experiments previously described. First, we checked which security controls were implemented by the apps. Next, we tried to bypass SSL pinning using the seven methods introduced above. Finally, we built a framework of applicability, relating the implemented security controls and the methods that are applicable.

5.1. SSL Pinning and Security Controls

In Table 4, we show the results of checking if the apps contain SSL pinning mechanisms. Moreover, we also checked whether the apps contain the security controls described in Section 3.

Table 4. Results of checking SSL pinning and controls.

Name App	SSL Pinning	Root Detection	Tamper Detection	Debug Detection	Obfuscated
Twitter	✓	-	-	-	✓
NAB Bank app	✓	✓	✓	✓	✓
Dropbox	✓	-	-	-	✓
WhatsApp	✓	✓	✓	✓	✓
Bank of America	✓	✓	✓	✓	✓
Facebook	✓	✓	✓	✓	-
Outlook	✓	✓	✓	-	-
PayPal	✓	-	-	-	-
MEGA	✓	-	-	-	-
Instagram	✓	✓	✓	✓	-

By analysing the results, the most outstanding facts are: (1) all apps implement SSL pinning mechanism; (2) six of the ten apps implement root detection and tamper detection mechanisms (in fact, they are the same ones); and (3) half the apps implement debug detection or obfuscation security controls. From the security point of view, we can consider Bank of America, NAB Bank app and

WhatsApp the most secure apps, since they implement all the security controls defined in Section 3. Facebook and Instagram are the second best, despite not implementing the code obfuscation security control. PayPal and MEGA only implement SSL pinning. The rest of the apps implement two or three security controls.

5.2. SSL Pinning Bypassing Methods

Table 5 provides the results of checking the SSL pinning bypassing methods. We use ✓symbol to indicate the correct bypassing and dash symbol otherwise. For the five old methods, we checked only the methods that appear as bypassed in Table 3, as we assume that the other methods are not applicable. On the other hand, for the new apps (i.e., Outlook and Instagram), all methods were checked.

By comparing the results for the old methods with the results in Table 5, there are very interesting facts to remark. The results for the iOS version of apps for Methods 1–3 remain equal. All apps were bypassed in the iOS versions using Methods 4 and 5. Nevertheless, only NAB Bank app, Bank of America and WhatsApp Android versions are not bypassed with Method 5. Those apps, together with Facebook and Instagram are not bypassed by Method 4.

Looking at the results for the new proposed methods, we can see how most of the apps that were vulnerable to the old methods are also bypassed with the new ones. There are other apps where the new methods fail to bypass SSL pinning, such as in the case of the Bank of America, NAB Bank app, Facebook, Instagram and WhatsApp.

In summary, we checked SSL bypassing with the five old methods, and we introduced two new bypassing methods.

Table 5. Results of bypassing SSL pinning methods.

App	**Principal Method**					**New Methods**	
	Method 1	**Method 2**	**Method 3**	**Method 4**	**Method 5**	**Method 6**	**Method 7**
Twitter	-	-	-	✓	✓	-	✓
NAB Bank app	-	-	-	-	-	-	-
Dropbox	-	-	✓	✓	✓	✓	✓
WhatsApp	-	-	-	-	-	-	-
Bank of America	-	-	-	-	-	-	-
Facebook	-	-	-	-	✓	-	-
Outlook	-	-	-	✓	✓	-	✓
PayPal	-	-	-	✓	✓	✓	✓
MEGA	-	-	-	✓	✓	✓	✓
Instagram	-	-	-	-	✓	-	-

5.3. Framework of Applicability

Based on the results shown in Table 5 and after analysing the experiments, we conclude that the framework of applicability is a key contribution of our research. This framework intends to help developers and auditors correctly develop and check secure communications, being a useful complement for the OWASP guidelines. Our framework helps simplifying the process of development and auditing Android mobile apps.

The framework of applicability is presented in Table 6. Each table cell indicates which SSL pinning methods are applicable depending on the implemented security control. Likewise, this table represents the impossibility of performing the SSL Pinning bypassing methods indicated by the $\rightleftharpoons$ symbol when the security controls in the first column are implemented. In the following, we explain in detail each entry of the table from the point of view of security controls:

- *Root detection* disables Methods 3 and 6 when it is implemented. This fact occurs as the device needs to be rooted in some of the steps taken in the application of the methods.
- *Anti-tampering* security control disables the application of Methods 2, 5 and 7. This fact occurs as it is necessary to modify statically the app code to bypass the SSL pinning checks. This step

requires to recompile the app and sign it. However, as we do not have access to the original certificate of the app, it is detected by the anti-tampering control.

- *Anti-debugging* security control stops Mmethods 3, 4, 6 and 7. As previously mentioned, a device in debug mode is detected, so the execution of the pinning is stopped since this measure stops step-by-step execution of the app code.
- *Code obfuscation* avoids Method 5 since obfuscation prevents the search of the functions that check the SSL Pinning.

On the other hand, we can read the table from point of view of the methods:

- *Method 2* can be tackled using the anti-tampering security control.
- *Method 3* can be stopped using root detection or anti-debugging controls.
- *Method 4* can be prevented with the anti-debugging control.
- *Method 5* can be stopped with the utilisation of anti-tampering or code obfuscation.
- *Method 6* can be held with the root detection and anti debugging security controls.
- *Method 7* can be denied with the anti-tampering and anti debugging controls.

This fact shows the need of implementing all the proposed security controls to stop all the SSL Pinning bypassing methods. It is remarkable that Method 1 is unrelated in the framework of applicability. Method 1 proves to be obsolete due to fact that it is easy to implement and most of the app developers may avoid it by changing the way of checking the certificate. Thus, Method 1 is, in practice, inoperative. In 2019, after the evolution in SSL pinning checking, changing CM in a certificate cannot be considered in SSL pinning bypassing, hence it is not applicable in any case. Moreover, we should point out that, even when a security control is not bypassed, we cannot guarantee that the control is completely secure. There is a certain possibility that a security control may be bypassed if the attacker has enough time. The auditor or the developer should evaluate the potential risk of SSL bypassing.

The framework of applicability provides a guideline with two main objectives: (1) learning the SSL pinning bypassing methods to check when some security controls are implemented; and (2) learning the security controls to implement against SSL pinning bypassing methods. Thus, the table can be read in two directions depending on the perspective to be used. For instance, if a pentester wants to check some security controls, the framework provides the SSL pinning bypassing methods to be used. On the other hand, if a developer wants an app to get protected against different SSL pinning bypassing methods, a set of security controls should be implemented.

In summary, the framework includes two dimensions: security controls and methods to circumvent SSL pinning. The first dimension describes the proposed methods to enforce the security of the SSL pinning mechanism from the point of view of the application developers. The second dimension deals with the methods to bypass SSL pinning depending on the scenario of application and the implemented controls. The second dimension is useful from the point of view of the auditors. The results of studying a set of apps regarding the security controls and methods returns the framework of applicability shown above. The experimentation supported the results given in Table 6. For instance, the cases of the apps Bank of America and WhatsApp demonstrate that the correct implementation of all security controls prevents the bypassing in any of the methods we applied. Otherwise, the incorrect application of security controls open a door to apply a set of methods to circumvent SSL pinning.

Table 6. Framework of applicability methods vs. security controls.

	Method 1	Method 2	Method 3	Method 4	Method 5	Method 6	Method 7
Root Detection			$\rightleftharpoons$			$\rightleftharpoons$	
Anti tampering		$\rightleftharpoons$			$\rightleftharpoons$		$\rightleftharpoons$
Anti debugging			$\rightleftharpoons$	$\rightleftharpoons$		$\rightleftharpoons$	$\rightleftharpoons$
Obfuscation					$\rightleftharpoons$		

6. Conclusions

Nowadays, the use of mobile technologies has grown exponentially. The market for mobile apps is producing a new level of security threats since most of the apps require communications with external services. In fact, insecure communication ranks #3 in OWASP Top 10, so it is also quite an important topic to be considered. Currently, the use of apps is at least as common as consulting Internet. It is crucial to secure these apps to guarantee the information confidentiality and integrity. In this respect, we provide to app developers and pentesters with a guideline to enforce security in the communications of Android apps.

In this paper, we show how SSL/TLS implementations are vulnerable even when using SSL pinning techniques. This way, some measures have to be taken to protect an app from SSL pinning bypassing. In the implementation phase of the app, root detection, debug detection and anti-tampering measures must be implemented. Finally, the code must also be obfuscated.

We also study the methods that can bypass these countermeasures. First, we study the current bypassing SSL pinning techniques. Bypassing techniques are quite well-known in the professional world, as a business line for security auditors and consultants, but it is unusual to see them in academia. Thus, we try to fill that research gap. The main contributions and value of this paper are the introduction of five known methods, the addition of two new methods for circumventing SSL pinning, the presentation of some solutions that developers can use to avoid this and the development of a framework of applicability. This framework can be useful as a guideline for both developers and pentesters.

Five methods are introduced: change CM in certificate, insert CA in the app, use SSL Unpinning, debugging app and modify app executable. Then, we propose two new methods to bypass SSL/TLS validations in Android devices. The first method deals with rooted devices and the second one deals with non-rooted devices. Finally, we evaluated our proposal, with the design of an experiment with a set of apps. The experiment consisted of two phases: (1) checking whether the app includes SSL pinning and contains any security controls; and (2) checking the seven methods presented in the paper. The set of apps we used consists of ten widely used applications in different fields.

The results demonstrate whether the Android version of the app can implement the same security controls as the iOS version. Moreover, the new bypassing methods can be useful as an addition to the older ones.

Finally, we propose a framework of applicability as a key contribution of our research. This framework intends to help developers and auditors for the correct development and checking of secure communications, being a useful complement for the OWASP guidelines. Our framework helps simplify the process of developing and auditing Android mobile apps. The framework of applicability provides a guideline with two main objectives: (1) learning the SSL pinning bypassing methods to check when some security controls are implemented; and (2) learning the security controls to implement against SSL pinning bypassing methods. This framework is useful both from the point of view of the developer and the security auditor. In addition, the framework of applicability shows the need of implementing all the proposed security controls to stop all the SSL Pinning bypassing methods. An evaluation of the framework of applicability will be part of our further research.

It has been demonstrated how SSL pinning can be circumvented, attacking the integrity of communications. We provide some mechanisms to prevent these attacks.

Author Contributions: F.J.R.-L. contributed in all phases of the paper; A.J.V.-V. conceived and design the experiments, analysed the results and wrote the paper; and J.R., J.L., and A.C. analysed the data and results and wrote the paper.

Funding: This work was funded by the Cátedra de Telefónica "Inteligencia en la red" of the Universidad de Sevilla and partially funded by the Ministry of Science and Technology of Spain through ECLIPSE (RTI2018-094283-B-C33), by the Junta de Andalucía via METAMORFOSIS projects, and the European Regional Development Fund (ERDF/FEDER).

Conflicts of Interest: All authors have approved the final content of the manuscript. No potential conflict of interest was reported by the authors.

Disclaimer: The experiments performed in this work were done strictly for academic purposes.

References

1. Li, D.; Guo, B.; Shen, Y.; Li, J.; Huang, Y. The evolution of open-source mobile applications: An empirical study. *J. Softw. Evol. Process.* **2017**, *29*, e1855. [CrossRef]
2. Unal, P.; Temizel, T.T.; Eren, P.E. What installed mobile applications tell about their owners and how they affect users' download behavior. *Telemat. Inform.* **2017**, *34*, 1153–1165. [CrossRef]
3. Kumar, R.; Perti, A. Security issues with self-signed SSL certificates. *Int. J. Innov. Technol. Explor. Eng. (IJITEE)* **2019**, *8*, 7S2.
4. Lindgren, A.; Lindoff, B. On Estimating the Number of Worldwide LTE Cell-IDs and WiFi Aps. 2018. Available online: https://combain.com/uploads/Whitepaper_WorldWide_LTE_CellID_and_WiFi_APs_A.pdf (accessed on 3 September 2019).
5. Anthi, E.; Theodorakopoulos, G. Sensitive data in Smartphone Applications: Where does it go? Can it be intercepted? In *International Conference on Security and Privacy in Communication Systems*; Springer: Berlin/Heidelberg, Germany, 2017; pp. 301–319.
6. Khan, J.; Abbas, H.; Al-Muhtadi, J. Survey on mobile user's data privacy threats and defense mechanisms. In Proceedings of the 12th Iberian Conference on Information Systems Technolo-Gies (CISTI), Lisbon, Portugal, 14–17 June 2017; No. 7975981.
7. D'Orazio, C.J.; Choo, K-K.R. A technique to circumvent SSL/TLS validations on iOS devices. *Future Gener. Comput. Syst.* **2017**, *74*, 366–374.
8. Razaghpanah, A.; Sundaresan, S.; Niaki, A.A, Amann, J.; Vallina-Rodriguez, N.; Gill, P. Studying TLS usage in Android apps. In Proceedings of the 13th International Conference on Emerging Technologies (CoNEXT 2017), Ingeon, Korea, 12–15 December 2017; pp. 350–362.
9. Fahl, S.; Harbach, M.; Perl, H.; Koetter, M.; Smith, M. Rethinking SSL development in an appified world. In Proceedings of the ACM SIGSAG Conference on Computer & Communications Security (CCS 2013), Berlin, Germany, 4–8 November 2013; pp. 49–60.
10. De los Santos, S.; Torres, J. Analysing HSTS and HPKP implementation in both browsers and servers. *IET Inf. Secur.* **2017**, *12*, 275–284.
11. Mueller, B.; Schleier, S. OWASP Mobile Application Security Verification Standard v 1.1.4. Available online: https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide (accessed on 21 November 2019). [CrossRef]
12. Dhawale, C.A.; Misra, S.; Jambhekar, N.D.; Thakur, S.U. Mobile computing security threats and solution. *Int. J. Pharm. Technol.* **2016**, *8*, 23075–23086.
13. OWASP Mobile Top 10. 2016. Available online: https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10 (accessed on 22 February 2017).
14. Kim, S.; Han, H.; Shin, D.; Jeun, I.; Jeong, H. A study of International Trend Analysis on Web Service Vulnerabilities in OWASP and WASC. In Proceedings of the 3rd International Conference on Information Security and Assurance (ISA 2009), Seoul, Korea, 25–27 June 2009; Springer: Heidelberg, Germany, Volume 5576, pp. 788–796.
15. Szczepanik, M.; Jozwiak, I. Security of mobile banking applications. *Adv. Intell. Syst. Comput.* **2018**, *635*, 412–419.
16. Hickman, K. *The SSL Protocol*. Netscape Communications Corp: Mountain View, CA, USA, 1995.
17. Dierks, T.; Rescorla, E. *The TLS Protocol Version 1.2*; RFC 5246. Available online: https://tools.ietf.org/html/rfc5246 (accessed on 21 November 2019).
18. Gu, X.; Gu, X. On the detection of fake certificates via attribute correlation. *Entropy* **2015**, *17*, 3806–3837.
19. Varela-Vaca, A.J.; Gasca, R.M. Towards the automatic and optimal selection of risk treatments for business processes using a constraint programming approach. *Inf. Softw. Technol.* **2013**, *55*, 1948–1973. [CrossRef]
20. Oracle—Java Secure Socket Extension (JSSE) Reference Guide. 2018. Available online: https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html (accessed on 3 September 2019). [CrossRef]

21. OpenSSL. Available online: https://www.openssl.org/ (accessed on 3 September 2019).
22. LibreSSL. Available online: http://www.libressl.org/ (accessed on 3 September 2019).
23. GNUTLS. Available online: https://www.gnutls.org/ (accessed on 3 September 2019).
24. Al-Qershi, F.; Al-Qurishi, M.; Md Mizanur Rahman, S.; Al-Amri, A. Android vs. iOS: The security battle. In Proceedings of the 2014 World Congress on Computer Applications and Information Systems (WCCAIS), Hammamet, Tunisia, 17–19 January 2014; pp. 1–8. doi:10.1109/WCCAIS.2014.6916629
25. Onwuzurike, L.; de Cristofaro, E. Danger is my middle name: experimenting with SSL vulnerabilities in Android apps. In Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks (WiSec '15), New York, NY, USA, 22–26 June 2015; ACM: New York, NY, USA, 2015. [CrossRef]
26. Fahl, S.; Harbach, M.; Muders, T.; Baumgärtner, L.; Freisleben, B.; Smith, M. Why eve and mallory love android: An analysis of android SSL (in)security. In Proceedings of the 2012 ACM conference on Computer and Communications Security (CCS '12), Raleigh, NC, USA, 16–18 October 2012; ACM: New York, NY, USA, 2012; pp. 50–61. [CrossRef]
27. Tendulkar, V.; Enck, W. An Application Package Configuration Approach to Mitigating Android SSL Vulnerabilities. *ArXiv* **2014**, arXv:1410.7745. [CrossRef]
28. Moonsamy, V.; Batten, L. Mitigating Man-In-the-Middle Attacks on Smartphones - A Discussion of SSL Pinning and DNSSec. In Proceedings of the 12th Australian Information Security Management Conference, Edith Cowan University, Perth, Australia, 2014.
29. Graves, J. SSL Pinning for Increased App Security. 2013. Available online: https://possiblemobile.com/2013/03/ssl-pinning-forincreased-app-security/ (accessed on 18 October 2019).
30. Andzakovic, D. Bypassing SSL Pinning on Android via Reverse Engineering. 2014. Available online: https://security-assessment.com/files/documents/whitepapers/BypassingSSLPinningonAndroidviaReverseEngineering.pdf (accessed on 16 September 2019).
31. Sierra, F.; Ramirez, A. Defending your android app. In Proceedings of the ACM Conference on Research in Information Technology, Chicago, IL, USA, 30 September–3 October 2015; pp. 29–34.
32. OWASP—Mobile Security Testing Guide—Android Anti-Reversing Defenses. Available online: https://mobile-security.gitbook.io/mobile-security-testing-guide/android-testing-guide/0x05j-testing-resiliency-against-reverse-engineering (accessed on 16 September 2019).
33. Apple Inc. Security Transforms Programming Guide. Available online: https://developer.apple.com/library/content/documentation/Security/Conceptual/SecTransformPG/SigningandVerifying/SigningandVerifying.html (accessed on 3 September 2019).
34. ProGuard. Available online: https://www.guardsquare.com/en/proguard (accessed on 3 September 2019).
35. iXGuard. Available online: https://www.guardsquare.com/en/ixguard (accessed on 3 September 2019).
36. APKtool. Available online: https://ibotpeaches.github.io/Apktool (accessed on 3 September 2019).
37. Penetration Testing Tool: dex2jar Package. Available online: https://tools.kali.org/reverse-engineering/dex2jar (accessed on 3 September 2019).
38. Android Developer: Logcat. logcat:https://developer.android.com/studio/command-line/logcat (accessed on 3 September 2019).
39. SSLUnpinning—Certificate Pinning Bypass. Available online: https://repo.xposed.info/module/mobi.acpm.sslunpinning (accessed on 16 September 2019).
40. Frida. Available online: https://www.frida.re/ (accessed on 3 September 2019).
41. Objection. Available online: https://github.com/sensepost/objection (accessed on 3 September 2019).