Guidelines Towards Secure SSL Pinning in Mobile Applications

F.J. Ramírez-López, A. J. Varela-Vaca, J. Ropero, A. Carrasco
Universidad de Sevilla, Spain
{framirez4, ajvarela, jropero, acarrasco}@us.es

***Resumen*—Security is a major concern in web applications for so long, but it is only recently that the use of mobile applications has reached the level of web services. This way, we are taking OWASP Top 10 Mobile as our starting point to secure mobile applications. Insecure communication is one of the most important topics to be considered. In fact, many mobile applications do not even implement SSL/TLS validations or may have SSL/TLS vulnerabilities. This paper explains how an application can be fortified using secure SSL pinning, and offers a three-step process as an improvement of OWASP Mobile recommendations to avoid SSL pinning bypassing. Therefore, following the process described in this paper, mobile application developers may establish a secure SSL/TLS communication.**

***Index Terms*—SSL pinning, security, mobile applications, certificate, OWASP**

Tipo de contribución: *Investigación original*

I. Introducción

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.

Every day, we read cases of users who have been scammed through the use of mobile applications [3]. For example, users may download some modified version of an application that is not controlled by the owner. In some cases, access to sensitive information from other users of the application has also been detected. This is due to the fact that many of the controls that have been applied in the web environment have not been considered in the mobile environment. Moreover, several mobile applications do not even implement SSL/TLS validations [4].

With this aim, the OWASP Mobile Application Security Verification Standard (MASVS) is an attempt to standardize these requirements using verification levels that fit different threat scenarios [5]. One of the most important challenges in mobile application security is to protect data flows over the insecure communication channel [6]. Insecure communications includes poor handshaking, incorrect SSL versions, weak negotiation or cleartext communication of Personally Identifiable Information (PII) [7]. Even when using SSL/TLS, applications may have vulnerabilities, especially to Man-in the-middle (MiTM) attacks [8]. Security measures such as SSL pinning are desirable [9]. Nevertheless, it is also possible to circumvent SSL/TLS validations [4]. In this paper, we explain how an application can be fortified taking into account certain security controls, where we should shield certain points to avoid attacks. This paper offers an improvement of OWASP mobile recommendations offering a three-step set of controls to avoid SSL pinning problems, and also wants to be a good practice guide for all the mobile application developers and users.

The paper is organized as follows. Section II deals with OWASP mobile recommendations. Section III introduces SSL validations and their possible vulnerabilities. Section IV offers solutions to SSL pinning bypass and shows a case of use. Finally, Section IV presents the conclusions of the paper.

II. OWASP Mobile recommendations

OWASP is the worldwide organization responsible for generating a standard for security in web applications [10]. 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 [11]. Based on our experience, we choose OWASP Top 10 Mobile as our starting point. Table I shows OWASP Mobile Top 10 in December 2016, which is the last update [7].

Tabla I
OWASP Top 10.

Category	Name
M1	Improper Platform Usage
M2	Insecure Data Storage
M3	Insecure Communication
M4	Insecure Authentication
M5	Insufficient Cryptography
M6	Insecure Authorization
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, so 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 applications may be reviewed, we are focusing on MASVS, which is defined in the OWASP Testing Guide. The OWASP Mobile Testing Guide has published recently its first version [5]. In this guide, security controls are defined and can be reviewed according to different categories. Every control describes the control itself, shows how can it be tested, and it sometimes offers a solution to the problem. However, the solution must be adapted to the system or the client that we are auditing.

Within the OWASP controls, there are several control layers that must be considered. The utilization of these layers depends on the application. There are three existing layers, called verification levels: L1, Standard Security; L2, Defense-in-Depth; and R, Resiliency Against Reverse Engineering and Tampering.

L1 layer controls - Standard Security. This control layer groups the most basic controls. These controls constitute a set of minimum characteristics that any application should accomplish. With these controls, a certain number of attacks on the application are avoided. This fact may be sufficient for some types of applications.

L2 layer controls - Defense-in-Depth. These controls are more advanced controls than the ones in the L1 layer. They help to avoid complex attacks on our application. This level is more demanding in terms of security since the controls ask for a more mature level of security in the application.

R layer controls - Resiliency Against Reverse Engineering and Tampering. This layer focuses on the reverse engineering attacks that can be done on an application. Therefore, it deals with everything that refers to both the code of an application and what can be modified within the source code. This constitutes an important level of verification of the source, hardware and other components in the application.

Depending on the type of application, several controls are used, while the others are discarded. For example, if an application only shows some information and no registration is needed, there is no point in applying L2 plus reverse engineering controls since there is not any sensitive information. The verification levels that applications may accomplish are the following:

- **MASVS-L1**. It constitutes the most basic security level, as there is no impact on the application development cost. All mobile apps must follow these requirements.
- **MASVS-L2**. E-Health and E-commerce applications, as they store sensitive PII.
- **MASVS-L1+R**. Applications with IP protections. Gaming industry.
- **MASVS-L2+R**. Applications managing critical data, like e-banking applications.

All the controls are grouped into categories, which are shown in Table II. In practice, category V1 is usually excluded, because those controls can be applied only in white-box tests or if we participate in the development of the application. As we can see, category V8 corresponds to level R.

Tabla II
CONTROL CATEGORIES.

Category	Name
V1	Architecture, Design and Threat Modelling Requirements
V2	Data Storage and Privacy Requirements
V3	Cryptography Requirements
V4	Authentication and Session Management Requirements
V5	Network Communication Requirements
V6	Platform Interaction Requirements
V7	Code Quality and Build Setting Requirements
V8	Resiliency Against Reverse Engineering Requirements

Within each category, we can distinguish which controls are applied at each level. We are focusing on category V5, Network Communication Requirements. To secure network communication, we should follow the recommendations shown in Table III.

Tabla III
NETWORK COMMUNICATION SECURITY VERIFICATION REQUIREMENTS.

Control	Description
5.1	Data is 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

In this paper, we show that it is only necessary to achieve the requirements corresponding to control 5.4. In practice, we can identify this control with SSL pinning.

III. WHY ARE SSL/TLS COMMUNICATIONS INSECURE?

Secure Socket Layer (SSL) [12] protocol and Transport Layer Security (TLS) [13] protocol, (hereinafter SSL/TLS) 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.

SSL/TSL protocol is based on a handshake sequence whose main features [14] 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; (4) Authenticate the client to the server.

There are several providers widely used as JSSE (Java Security Socket Extension) [15], OpenSSL [16], LibreSSL [17], or GnuTLS [18]. Even there exist specific hardware with built-in SSL/TLS solutions such as iOS devices.

III-A. SSL/TLS vulnerabilities: Bypassing SSL/TLS

As mentioned before, one of the top-3 risks identified by OWASP is an insecure communication due to a poor configuration of an SSL/TLS channel. However, SSL/TLS is a non-free vulnerability protocol since it can be broken by MiTM attacks [11]. MiTM attacks take place due to lack of validation or incorrect validation in the protocol.

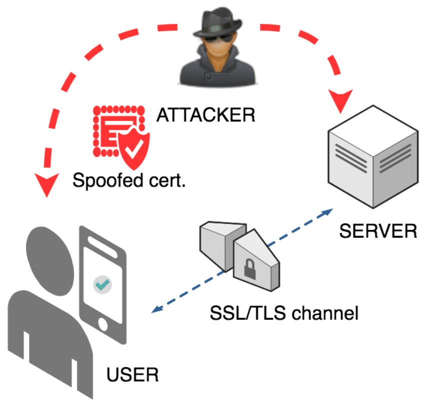

Figura 1. Bypassing SSL/TLS by using spoofed certificates.

In SSL/TLS, certificates are verified to check whether they are signed by proper CA. In this case, we can mislead the application giving a certificate (cf. spoofed certificated, see Fig. 1). The certificate is trusted, 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 decrypting all the packets in the communication.

III-B. Solution to Bypassing SSL/TLS: SSL pinning

The pinning technique or HTTP Public Key Pinning (HPKP) [4] has emerged in the last years as a security control to fortified HTTPS-based applications against MiTM attacks.

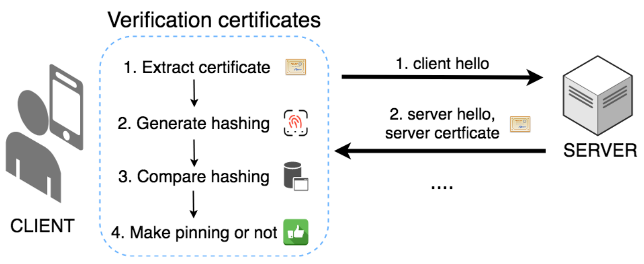

Figura 2. Certificate verification process and pinning.

Fig. 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 (cf. server hello in Fig. 2). Then, the client asks for the server's certification when server answers with the content of the information of its certificate and public key (cf. verification certificates in Fig. 2). The second stage is called pinning. The mobile device follows a verification process where the certificate is received from the server. Besides, the 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.

III-C. Vulnerabilities of SSL Pinning: Bypassing SSL pinning

SSL pinning is also vulnerable when it is not well implemented. There are several ways to bypassing it, as described by D'Orazio and Choo [4] or by Andzakovic [19]. Several tools can be used to bypassing SSL pinning, such as follows:

- *SSL Kill Switch 2* takes advantage of the fact that the code that implements SSL Pinning is a known code. Application developers use a well-known or common template. In this case, an attacker may guess this and use SSL Kill Switch 2 to bypass SSL Pinning in the application.
- *Dynamic analysis of code* can be applied. For example, Frida or Cycript enable the modification of some functions of the application in runtime.
- If the application does not implement anti-tampering or exceptions for the modification of the application, SSL Pinning functions can be replaced to bypass the pinning process.

IV. Good practices to implement Secure SSL Pinning

Here, we present some good practices or guidelines as a set of several steps to implement an adequate secure solution to the SSL Pinning. This way, bypassing SSL Pinning is avoided. Although, OWASP propose to use a set of controls in order to ensure channels of communications, our guideline demonstrates and ensures that with only three steps the mobile applications can be fortified against bypassing SSL pin-ning and no more control need to be checked.

The proposed process is shown Fig. 3 and indicates the points that have to be tackled to solve the problems mentioned in the previous section. All the measures that should be taken are described below.

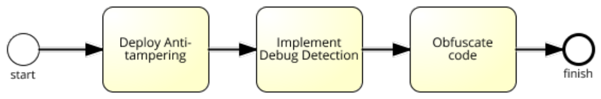

Figura 3. Process to ensure SSL Pinning.

1. **Deploy Anti-tampering** solution is an important option since any attacker may decompile the application and recompile it. Modifying some parts of our application, as previously explained in the SSL pinning bypass process, an attacker could skip the SSL pinning, and thus make our application invalid.
   Listing 1 gives an example of the code that can be included into an Android application, particularly in the *onCreate* function within *MainActivity*. so that nothing else but starting check the signature of the application. In iOS, the mechanism is similar, as indicated in the Apple security transforms programming guide [20].

```
for (Signature signature : packageInfo.signatures) {
  byte[] signatureBytes = signature.toByteArray();
  MessageDigest md =
      MessageDigest.getInstance("SHA");
  md.update(signature.toByteArray());
  final String currentSignature =
      Base64.encodeToSpring(md.digest(),
    Base64.DEFAULT);
  Log.d("REMOVE\ME","Include this string as a value
      for SIGNATURE:" +
    currentSignature);
  //compare signatures
  if (SIGNATURE.equals (currentSignature)){
```

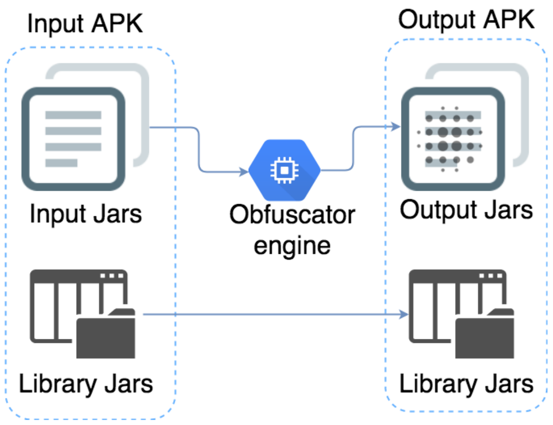

Figura 4. Obfuscation process.

```
    return VALID;
    };
}
return INVALID;
}
...
```

Listing 1. Example of code to check signatures.

2. **Implement Debug Detection**, to prevent our code from being controlled. If we detect that our device is in debug mode, the execution of the application is stopped. This measure stops an attacker from seeing our code step-by-step behavior. Together with next measure, obfuscation of code, allows hiding the internal functioning of our application. We may use functions as the one we are providing in listing 2 for our Android application.

```
protected void OnCreate (Bundle bundle) {
  if (sg.vantagepoint.a.c.a() ||
      sg.vantagepoint.a.c.b()
    || sg.vantagepoint.a.c.c()){
    this.a ("Root detected!"); //This is the message
        we are looking for
  }

  if (sg.antagepoint.a.b.a((Context)this.
          getApplicationContext())) {
    this.a("App is debuggable!");
  }
  super.onCreate(bundle);
  this.setContentView(2130903040);
}
```

Listing 2. Example of code to avoid debugging mode.

3. **Obfuscate code**. All the measures above do not make sense without prevent-ing any attacker from knowing our code and making the analysis of SSL Pinning functions or any of the previous ones more difficult. For this reason, code obfuscation is necessary. There are code obfuscators which convert our existing code into a more illegible code (cf. Fig. 4). Therefore, it is more difficult to detect which are the critical functions of the code for an attacker.
   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 listing 3, where names of variables and functions are hidden.

```
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 {
}
```

Listing 3. Example of obfuscation code.

There are numerous tools on Android and iOS that allow the obfuscation of code, as *Proguard* [21] for Android or *iXGuard* [22] for iOS.

IV-A. Case of use: securing a mobile application

The analysis carried out in [4], where 40 mobile applications from different environments were analyzed, showed that only 10 of the applications used SSL pinning. Moreover, all these applications are vulnerable when tampering with application at runtime, or when modifying the application executable. Next, we offer a solution that fixes all these vulnerabilities, but focusing on an Android mobile application case study.

Samsung Galaxy J5 device and Android 8.0 Oreo OS have been used to carry out the tests detailed in this section. Regarding the application, we have customized an Android-based template [16], but all the results are also applicable to iOS systems. Many functions are given in the GitHub AeroGear library [23]. The template is given by default with a set of tests to check some security controls, such as root detection, device lock, etc. These controls are related to the ones proposed by OWASP mobile recommendations. However, other security controls such as anti-tampering are not included. In order to illustrate the application and the effectiveness of our guideline, the three-step process is detailed below:

Step 1. Detection of the debug mode in the application. We must verify some controls of our device, as the detection of debug mode, hooking tools and emulation mode. Debug and emulation mode can be detected by means of the code shown in listing 4.

```
public void debuggerDetected() {
  totalTests++;
  SecurityCheckResult result = securitySer-vuce.check(
      SecurityCheckType.IS_DEBUGGER);
  if (result.passed()) {
    setDetected(debuggerAccess,
      R.string.debugger_detected_positive);
  }
}

public void detectEmulator() {
totalTests++;
  SecurityCheckResult result = securitySer-vuce.check(
      SecurityCheckType.IS_EMULATOR);
  if (result.passed()) {
    setDetected(emulatorAccess,
      R.string.emulator_detected_positive);
  }
}
```

Listing 4. Example of code to avoid debugging mode.

The function *detectEmulator* call another function inside it, named *securityService.check*. This function depends on the AeroGear library, where we can find the function shown in listing 5. This function checks if there is any debugger connected to the device.

```
protected boolean execute(@NonNull Context context){
  return !Debug.isDebuggerConnected();
}
```

Listing 5. Function checking if the debugger is connected.

We can also provide a function for detecting Hooking tools. This function is important to prevent tools like Xposed (i.e., JustTrustMe and SSLUnpinning 2.0 modules) from using a process to bypass SSL pinning. It is not exactly a debugging process, but is it is quite similar, as we are debugging the process in memory several times. Listing **??** shows Hooking tool detection function.

```
public void detectHookingFramework() {
  totalTests++;
  String xposedPackageName =
      "de.robv.android.xposed.installer";
  String substratePackageName = "com.saurik.substrate";
  if (checkAppInstalled(xposedPackageName) ||
    checkAppIn-stalled(substratePackageName))
  {
  setDetected(hookingDetected,
      R.string.hooking_detected_positive);
  }}
}
```

Listing 6. Function for detecting Hooking tools.

This way, the application cannot be debugged, as it would detect the debug mode, as shown in Fig. 5 Thus, it is impossible to circumvent SSL pinning tampering with application at runtime. This is due to the fact that all the frameworks used with this aim cannot be directly used.

Step 2. Check of the anti-tampering solution. The used template uncovers anti-tampering control. Thus, it must be added inside the method *checkAppInstalled*, which checks if the application was downloaded from a correct source. The proposed code is highlighted in red in listing 7.

```
@RequiresApi(api = Build.VERSION_CODES.M)
public void detectAntiTampering() throws
     PackageManager.NameNotFoundException,
     No-SuchAlgorithmException {
boolean result = true;
String packageName =
     "com.feedhenry.securenativeandroidtemplate";
PackageManager packageManager =
     this.getContext().getPackageManager();
packageManager.getPackageInfo (packageName, 0);
for (android.content.pm.Signature signature :
packageManager.getPackageInfo(packageName, 0).signatures)
{
byte[] signatureBytes = signature.toByteArray ();
MessageDigest md = MessageDigest.getInstance ( "SHA" );
md.update ( signature.toByteArray () );
final String currentSignature = Base64.encodeToString (
     md.digest (),
Base64.DEFAULT );
//compare signatures
if ("478yYkKAQF+KST8y4ATKvHkYibo".equals (
     currentSignature )) {
result = true;
}
}

if (!result) {
WarningDialog warning = WarningDialog.createWarningDialog (
"Application is not original");
warning.show(getFragmentManager(), "device_warning");
}
}
}
```

Listing 7. New code to check anti-tampering.

With this code, APK (Android Application Package) can be modified, and compiled again. The application then warns about the existence of an error in the signature verification, as shown in Fig. 6. This step prevents attackers from SSL pinning bypassing, as the attackers should also bypass the modifications.

Step 3. Code obfuscation. Code obfuscation is done just to hide SSL pinning methods, and the verifications mentioned above. This step makes it difficult for the attacker to check the source code. This way, we are preventing any bypassing method. The obfuscation may be configured in the setup of the project using a third-party library, as mentioned previously. The templated is already prepared to use obfuscation, and it is preconfigured to use Proguard with this aim. Listing 8 the obfuscation code. The obfuscation code may be configured in the gradle of the application. The functioning is similar for other packaging systems, like iOS systems.

```
release {
  versionNameSuffux System.getenv("CIRCLE_BUILD_NUM")
  signingConfig signingConfigs.release
  minifyEnabled true
  proguardFiles.getDefaultProguardFile(
    'proguard-android.txt'),
    'proguard-rules.pro'
}
```

Listing 8. Obfuscation code configuration.

We can check Proguard configuration opening the files mentioned as in listing 8. Moreover, a piece of code of the resulting configuration might be as shown in listing 9. In this code, the two first lines are referenced to the used obfuscation tools. The third line is used to erase the logs.

```
  -dontwarn com.google.errorprone.annotations.*
  -dontwarn okio.**
  -dontwarn org.slf4j.**
```

Listing 9. Proguard configuration.

After obfuscation is done, the code might be decompiled from the APK. However, the code is complete unpredictable, and it is impossible to determine which functions are responsible for the verification of the debug mode detection. Listing 10 shows an example of obfuscated code from the functions of listing 7.

```
public abstract class b implements a {
  private Fragment a;
  private c b;

  public b(Fragment fragment) {
    this.a = fragment;
    this.b = new c();
  }

  private Context d() {
    return this.a.getActivity();
  }

  public void a() {
    Context d = d();
    if (d != null) {
      this.b.a(d);
```

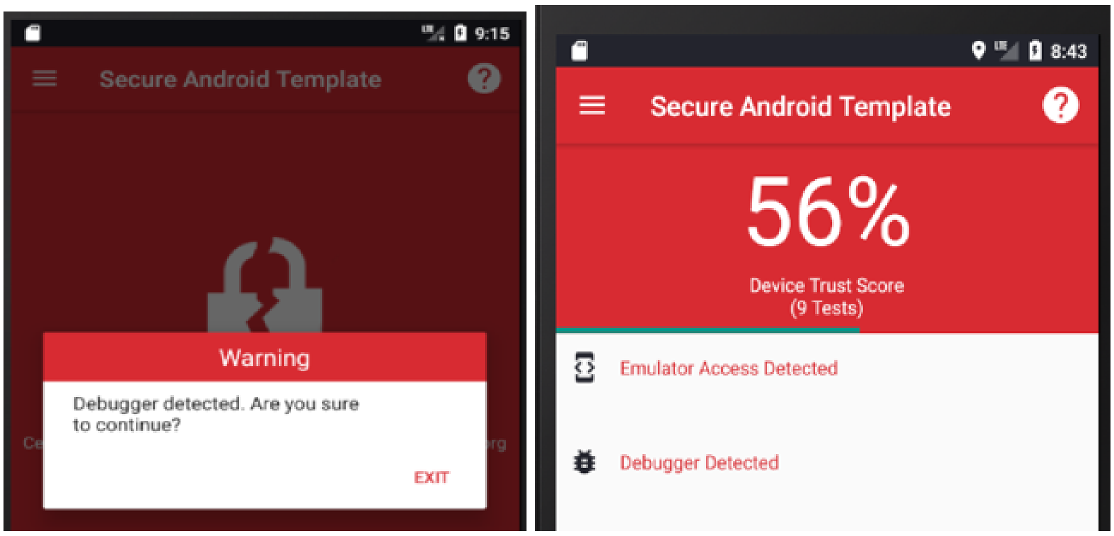

Figura 5. Results debug detection in the template and device trust score.

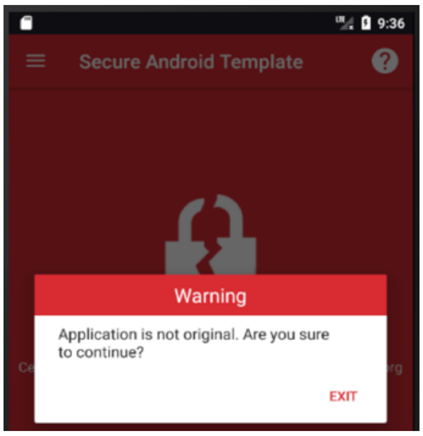

Figura 6. Results of the Application after checking application originality.

```
        }
    }

    ...
}
```

Listing 10. Obfuscated code extracted.

With these three steps, all the methods to bypass SSL pinning are fully invalid, so that a more secure communication channel is created, and other controls do not need to be checked.

V. Conclusions

This paper presents some guidelines for implementing secure communications. We offer solutions to SSL pinning problems, introducing some good practices for all the mobile application developers and users.

SSL/TLS provides confidentiality, message integrity, and authentication in data communication. However, SSL/TLS is a non-free vulnerability protocol since it can be broken by MiTM attacks. The pinning technique has emerged in the last years as a security control to fortified applications against MiTM attacks. SSL pinning is also vulnerable when it is not well implemented. There are several ways to circumventing it, like using SSL Kill Switch 2, a dynamic analysis of the application code, or not implementing anti-tampering.

We propose a securing mechanism that implements three security measures. First, an anti-tampering solution must be deployed, modifying some parts of the application. Second, it is necessary to implement debug detection, to prevent the application code from being controlled. Finally, obfuscating code converts our existing code into a more illegible code. This way, SSL/TLS is totally secure.

Concluding, we used an Android-based templated mobile application to implement the proposed measures, and we demonstrated that it is converted into a secure applica-tion using the SSL Pinning mechanism.

As future lines, we would like to test the proposed mecha-nism with more applications, to prove the universality of the method.

Acknowledgement

This work has been partially funded by the Ministry of Science and Technology of Spain through ECLIPSE (RTI2018-094283-B-C33), the Junta de Andalucia via the PIRAMIDE and METAMORFOSIS projects, the European Regional Development Fund (ERDF/FEDER). The authors would like to thank the Cátedra de Telefónica “Inteligencia en la red“ of the Universidad de Sevilla for its support.

Referencias

[1] Li, D., Guo, B., Shen, Y., Li, J., Huang, Y.: The evolution of open-source mobile appli-cations: An empirical study. Journal of Software: Evolution and Process 29 (7), Article number e1855 (2017).
[2] Unal, P., Temizel, T.T., Eren, P.E.: What installed mobile applications tell about their owners and how they affect users' download behavior. Telematics and Informatics 34 (7), 1153-1165 (2017).
[3] Khan, J., Abbas, H., Al-Muhtadi, J. Survey on mobile user's data privacy threats and defense mechanisms. In: 12th Iberian Conference on Information Systems Technolo-gies, CISTI, article number 7975981, Lisbon, Portugal (2017).
[4] D'Orazio, C.J., Choo, K-K.R. A technique to circumvent SSL/TLS validations on iOS devices. Future Generation Computer Systems 74, 366-374 (2017).
[5] Mueller, B., Schleier, S. OWASP Mobile Application Security Verifica-tion Standard v 1.0. Last Consulted: March 2018.

[6] Dhawale, C.A., Misra, S., Jambhekar, N.D., Thakur, S.U. Mobile computing security threats and solution. International Journal of Pharmacy and Technology 8 (4), 23075-23086 (2016).
[7] OWASP Mobile Top 10 2016. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10. Last modified: February 2017.
[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 con-ference on emerging technologies, CoNEXT 2017, pp. 350-362, Ingeon, South Korea (2017).
[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, pp. 49-60, Berlin, Germany, 2013.
[10] 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: 3rd International Confer-ence on Information Security and Assurance, ISA 2009, LNCS, vol. 5576, pp. 788-796. Springer, Heidelberg (2009).
[11] Szczepanik, M., Jozwiak, I. Security of mobile banking applications. Advances in In-telligent Systems and Computing 635, 412-419 (2018).
[12] Hickman, K. The SSL Protocol. Netscape Communications Corp (1995).
[13] Dierks, T., Rescorla, E. The TLS Protocol Version 1.2. RFC 5246 (2008).
[14] Varela-Vaca, A.J., Gasca, R.M. Towards the automatic and optimal selection of risk treatments for business processes using a constraint programming approach. Information & software technology, vol. 55(11), pp. 1948-1973 (2013).
[15] Oracle – Java Secure Socket Extension (JSSE) Reference Guide. https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html (2018). Last consulted: April 2018.
[16] OpenSSL. https://www.openssl.org/. Last consulted: April 2018.
[17] LibreSSL. http://www.libressl.org/. Last consulted: April 2018.
[18] GNUTLS. https://www.gnutls.org/. Last consulted: April 2018.
[19] Andzakovic, D. Bypassing SSL Pinning on Android via Reverse Engineering. https://security-assessment.com/files/documents/whitepapers/Bypassing%20SSL%20Pinning%20on%20Android%20via%20Reverse%20Engineering.pdf. Last consulted: March 2018.
[20] Apple Inc. Security Transforms Programming Guide. https://developer.apple.com/library/content/documentation/Security/Conceptual/SecTransformPG/SigningandVerifying/SigningandVerifying.html. Last consulted: March 2018.
[21] ProGuard. https://www.guardsquare.com/en/proguard. Last consulted: April 2018.
[22] iXGuard. https://www.guardsquare.com/en/ixguard. Last consulted: April 2018.
[23] AeroGear Services Android SDK. https://github.com/aerogear/aerogear-android-sdk. Last consulted: April 2018.
[24] FeedHenry Templates – RedHat, http://feedhenry.org/. Last consulted: April, 2018.