CMP417 - Engineering Resilient Systems 1

Software Security

Jack Bowker – 1803838

March 2021

1. ABSTRACT

This report is a short look at a secure software development technique, aimed at a small company whose IT Infrastructure is being targeted by a Hacktivist group. The introduction covers why it's so important to integrate best security practice at the core of development, as threats worsen. The background section focuses on the vulnerability used to demonstrate the risk to the company, specifically a CWE relating to Insecure Communication. This is used to underscore the possible risk if sensitive data is sent over an insecure protocol. The recommendation section covers how the company can overcome these vulnerabilities using secure code review techniques, giving context on what resources are available. The implementation section will detail a specific CVE in a mobile application that doesn't use secure communication methods, giving examples to how the recommended techniques would have prevented this vulnerability. The Limitations & Challenges section covers the benefits and weaknesses of the individual techniques and how they should be part of a holistic approach.

2. Introduction

Adopting and following secure software development processes is more important than ever, with the number of targeted breaches in the US growing by 27% yearly (PurpleSec, 2020). By sticking to best practice using a thorough secure development pipeline, the risk of publishing insecure or vulnerable code is reduced significantly. Ideally, good security practice should be proactive and at the core of all software development. In reality, it often only comes after a breach scares companies into reacting, as according to McAfee, the average organization has at least 14 misconfigured IaaS (Infrastructure as a Service) instances running at any given time (McAfee, 2019). A possible reason for this is that until a breach occurs, a good security posture can be hard to justify if budgets are tight. It can also appear to conflict with other concerns, such as performance and development time. In reality however, according to the Systems Sciences Institute it costs six times more to fix a bug found during implementation versus during design, as shown in Figure 1.

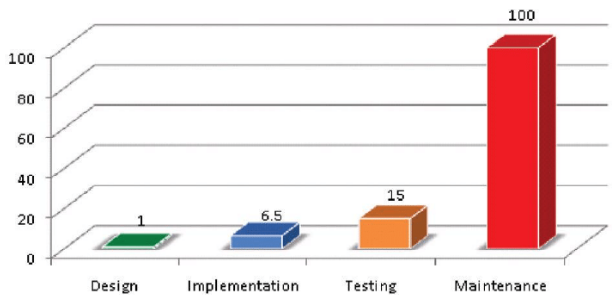


Figure 1 - Relative cost of fixing defects (Dawson et al., 2010)

The best practice is to integrate security testing into all stages of development including when planning, designing, coding, evaluating, and continuing this after the release. Shown in Figure 2 is a demonstration of what a thorough Secure Development Life Cycle (SDLC) could look like.

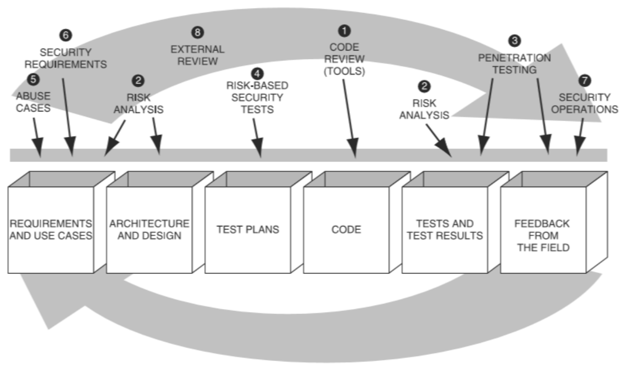


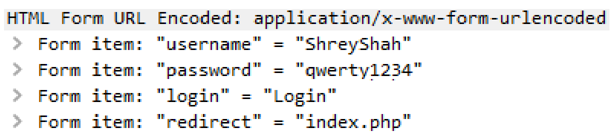
Figure 2 - Secure Software Development Life Cycle Process (Hudaib et al., 2017)

3. Background

After being targeted by a Hacktivist group, a security review was carried out on the small company's IT Infrastructure. According to the brief, the mobile application provided is a customized system that allows staff to change their own account details. This most likely necessitates sensitive personal information and login data being transferred between the mobile device and a remote database. If not secured properly, this traffic could be easily intercepted using techniques such as a Man in the Middle (MITM) attack on the outgoing traffic if a staff member were to connect to an insecure Wi-Fi network, for example.

This vulnerability would be classed as 'Missing Encryption of Sensitive Data' (311) in Mitre's Common Weakness Enumeration (CWE) database (MITRE, no date). This CWE is a broad description, with related weakness including 'Protection Mechanism Failure' and 'Cleartext Transmission of Sensitive Information'.

According to OWASP's Top 10 Mobile Risks (*OWASP Mobile Top 10*, 2016), 'Insecure Communication' is the third listed security risk when developing mobile applications, due in part to how easy it is for malicious actors to exploit with freely available tools such as WireShark, along with the potential harm done if sensitive data such as login details are leaked. Shown below in Figure 3 is an example of how easily unencrypted traffic can be captured.



```
HTML Form URL Encoded: application/x-www-form-urlencoded
> Form item: "username" = "ShreyShah"
> Form item: "password" = "qwerty1234"
> Form item: "login" = "Login"
> Form item: "redirect" = "index.php"
```

Figure 3 - WireShark capture of unencrypted HTTP traffic from PHP form

4. Recommendation

An effective secure development practice that could help catch vulnerabilities like insecure communication would be to carry out a thorough secure code review, using both automated analysis techniques and manual review.

Static analysis tools can be used to carry out 'Prefix analysis', which approximates what should be at the start of a string to detect what protocols are used in application URLs (Ferrara *et al.*, 2021), for example, a vulnerable application may point at an API endpoint that starts with 'http://'. Similarly, static analysis can check for code that's hardcoded to communicate via TCP Port 80, traditionally used by HTTP traffic.

Dynamic techniques have also proven effective, with the automated 'AndroShield' tool using a hybrid approach to monitor network traffic and application behaviour while the application is running. The tool checks for insecure network requests, information leaks, as well as crashes and other possible vulnerabilities (Amin *et al.*, 2019).

Along with integrating automated tools into the development process, a manual review of the existing application should be carried out, with new code analysed regularly to prevent future vulnerabilities going unchecked. The OWASP Foundation provides a thorough code review guide that covers relevant sections, with Section A6 – Sensitive Data Exposure (OWASP Foundation, 2017) aimed at web applications but still relevant to the mobile application. It states that when dealing with sensitive data to use Validated Implementations of cryptographic functions, along with a list of items to look out for including using TLS for any login/authenticated pages, disabling insecure HTTP interfaces, and good practice for storing authentication cookies.

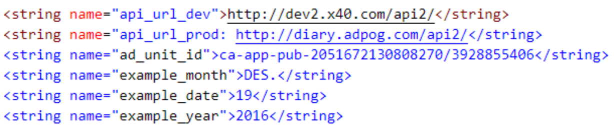
5. Implementation

It was found that the company's mobile application sent data insecurely, as stated in CWE 311 in Mitre's Common Weakness Enumeration list. Multiple existing CVE (Common Vulnerabilities and Exposures) listings for similar vulnerabilities in mobile applications were found.

"Diary with lock" Android App (MITRE, 2017)

cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15581

This CVE relates to the free Android application "Diary with lock", with version 4.7.2 being targeted specifically. It was found to be transmitting both login data and diary entries without HTTPS or any other encryption, whilst stating the diary was intended to store "secrets". The APK was decompiled using JADX (skylot, 2021), and after a first look it was confirmed that "strings.xml" pointed to an insecure API endpoint as shown below in Figure 4.

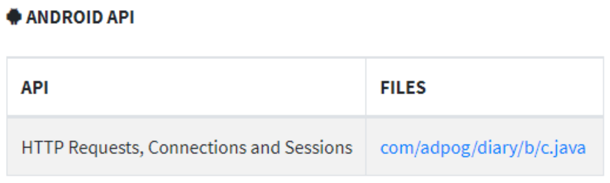


```
<string name="api_url_dev">http://dev2.x40.com/api2/</string>
<string name="api_url_prod: http://diary.adpog.com/api2/</string>
<string name="ad_unit_id">ca-app-pub-2051672130808270/3928855406</string>
<string name="example_month">DES.</string>
<string name="example_date">19</string>
<string name="example_year">2016</string>
```

Figure 4 - API endpoint for Diary app showing HTTP endpoint

As well as being sent to a HTTP endpoint, it was found that no other encryption was being done on the traffic being sent. Implementing a secure code review would have prevented this vulnerability from occurring, with an array of tools and guides to assist.

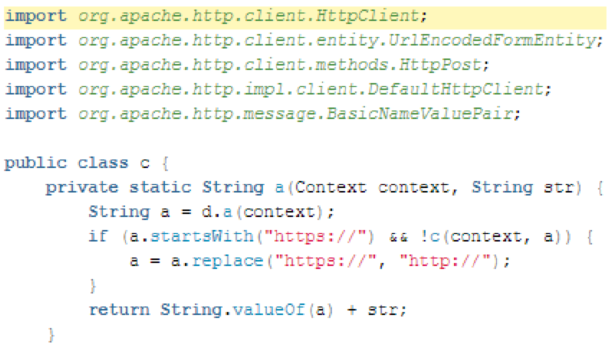
The first tool used was the open source Mobile Security Framework (*MobSF/Mobile-Security-Framework-MobSF*, 2021) tool, which detected HTTP requests using static analysis and pointed to relevant points in the application's source code, as shown in Figure 5 and Figure 6.



ANDROID API

API	FILES
HTTP Requests, Connections and Sessions	com/adpog/diary/b/c.java

Figure 5 - MobSF tool detecting HTTP API usage

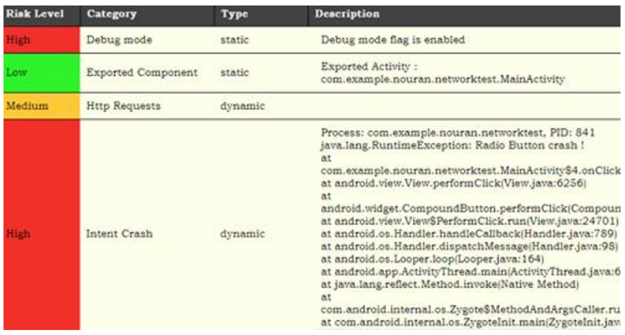


```
import org.apache.http.client.HttpClient;
import org.apache.http.client.entity.UrlEncodedFormEntity;
import org.apache.http.client.methods.HttpPost;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.message.BasicNameValuePair;

public class c {
    private static String a(Context context, String str) {
        String a = d.a(context);
        if (a.startsWith("https://") && !c(context, a)) {
            a = a.replace("https://", "http://");
        }
        return String.valueOf(a) + str;
    }
```

Figure 6 - Point in code MobSF directed to, with unusual string replacement of HTTPS

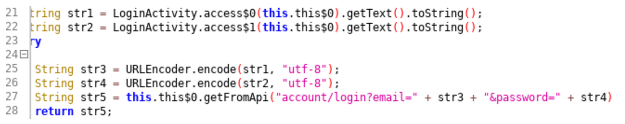
Due to not being able to run the "Diary with lock" Android application directly, dynamic analysis couldn't be carried out but shown below in Figure 7 is an example of the AndroShield open source tool detecting insecure requests by monitoring the network traffic of the application as it runs.



Risk Level	Category	Type	Description
High	Debug mode	static	Debug mode flag is enabled
Low	Exported Component	static	Exported Activity : com.example.nouran.networktest.MainActivity
Medium	Http Requests	dynamic	
High	Intent Crash	dynamic	Process: com.example.nouran.networktest, PID: 841 java.lang.RuntimeException: Radio Button crash ! at com.example.nouran.networktest.MainActivity$4.onClick at android.view.View.performClick(View.java:6256) at android.widget.CompoundButton.performClick(Compoun at android.view.View$PerformClick.run(View.java:24701) at android.os.Handler.handleCallback(Handler.java:789) at android.os.Handler.dispatchMessage(Handler.java:98) at android.os.Looper.loop(Looper.java:164) at android.app.ActivityThread.main(ActivityThread.java:6 at java.lang.reflect.Method.invoke(Native Method) at com.android.internal.os.Zygote$MethodAndArgsCaller.ru at com.android.internal.os.ZygoteInit.main(ZygoteInit.jav

Figure 7 - Vulnerabilities report from AndroShield tool, showing detected HTTP requests (Amin et al., 2019)

As shown in Figure 8 inside the Diary app's LoginActivity source code, the user's email and password are passed directly into the API query meaning email addresses, passwords and diary entries were sent with no encryption.



```
tring str1 = LoginActivity.access$0(this.this$0).getText().toString();
tring str2 = LoginActivity.access$1(this.this$0).getText().toString();
ry

 String str3 = URLEncoder.encode(str1, "utf-8");
 String str4 = URLEncoder.encode(str2, "utf-8");
 String str5 = this.this$0.getFromApi("account/login?email=" + str3 + "&password=" + str4)
 return str5;
```

Figure 8 - LoginActivity from Diary app showing no encryption on password ('Auditing WriteDiary.com', 2017)

A manual code review working from a respected source such as OWASP's Code Review Guide (OWASP Foundation, 2017) would have also prevented this code from being shipped, as in Section A6/12.3 it clearly advises against putting "sensitive data in the URL" through a GET request for example, as well as suggesting that TLS should be used for all connections ideally. Synk's Best Practice list (Vermeer and Gee, 2020) states similarly that "Obviously sending somebody's credit card details as a query parameter or as plain text in the payload over HTTP is not considered safe at all."

6. Limitations & Challenges

Implementing secure code review with a multipronged approach including static & dynamic analysis and manual code review can be an effective way to catch more vulnerable code before it's published, however there are still limitations to what using these techniques solely can accomplish.

Static analysis is a low effort way to find obvious coding errors early in the development stage, however it can often be hard to find legitimate vulnerabilities. This is due to the relatively noisy output static analysis generates that often includes numerous false positives, due to the rule-based scanning technique that can be incorrect in certain situations. (*Static Code Analysis Control | OWASP Foundation*, no date) As well as this, due to analysing source code, static analysis can't help in finding runtime vulnerabilities.

Dynamic analysis tools can often find vulnerabilities that static analysis can't, due to not needing access to source code as well as the technique of monitoring of the application at runtime. In the case of the "Diary with lock" app it can be used to monitor the network to confirm if it's secure. This technique however can provide a false sense of security if automated tools don't find any vulnerabilities, as well as identifying false positives and negatives.

Manual code review is invaluable especially if the development team follows similar practices and syntax. It's most effective if used in combination with static/dynamic analysis techniques to analyse results further, with proper context of what the code is supposed to do. The biggest downside of manual review especially if developers don't have good knowledge of the code, is that it's very time consuming and can be hard to justify if budgets or deadlines are tight.

7. References

Amin, A. *et al.* (2019) 'AndroShield: Automated Android Applications Vulnerability Detection, a Hybrid Static and Dynamic Analysis Approach', *Information*, 10(10), p. 326. doi: 10.3390/info10100326.

'Auditing WriteDiary.com' (2017) *Auditing WriteDiary.com (CVE-2017-15581 & CVE-2017-15582)*, 27 October. Available at: https://1337sec.blogspot.com/2017/10/auditing-writediarycom-cve-2017-15581.html (Accessed: 14 March 2021).

Dawson, M. *et al.* (2010) 'Integrating Software Assurance into the Software Development Life Cycle (SDLC)', *Journal of Information Systems Technology and Planning*, 3, pp. 49–53.

Ferrara, P. *et al.* (2021) 'Static analysis for discovering IoT vulnerabilities', *International Journal on Software Tools for Technology Transfer*, 23(1), pp. 71–88. doi: 10.1007/s10009-020-00592-x.

Hudaib, A. *et al.* (2017) 'A Survey on Design Methods for Secure Software Development', *International Journal of Computers and Technology*, 16, pp. 7047–7064. doi: 10.24297/ijct.v16i7.6467.

McAfee (2019) 'Cloud Adoption and Risk Report', p. 20.

MITRE (2017) *CVE - CVE-2017-15581*. Available at: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-15581 (Accessed: 15 March 2021).

MITRE (no date) *CWE-311: Missing Encryption of Sensitive Data (4.3)*. Available at: https://cwe.mitre.org/data/definitions/311.html (Accessed: 12 March 2021).

MobSF/Mobile-Security-Framework-MobSF (2021). Mobile Security Framework. Available at: https://github.com/MobSF/Mobile-Security-Framework-MobSF (Accessed: 15 March 2021).

OWASP Foundation (2017) *OWASP Code Review Guide*. Available at: https://owasp.org/www-pdf-archive/OWASP_Code_Review_Guide_v2.pdf (Accessed: 14 March 2021).

OWASP Mobile Top 10 (2016). Available at: https://owasp.org/www-project-mobile-top-10/ (Accessed: 11 March 2021).

PurpleSec (2020) '2019 Cyber Security Statistics Trends & Data', *PurpleSec*, 8 November. Available at: https://purplesec.us/resources/cyber-security-statistics/ (Accessed: 11 March 2021).

skylot (2021) *skylot/jadx*. Available at: https://github.com/skylot/jadx (Accessed: 15 March 2021).

Static Code Analysis Control | OWASP Foundation (no date). Available at: https://owasp.org/www-community/controls/Static_Code_Analysis (Accessed: 15 March 2021).

Vermeer, B. and Gee, T. (2020) *Secure Code Review | Snyk*. Available at: https://snyk.io/blog/secure-code-review/ (Accessed: 15 March 2021).