CSCI 455: Analysis Plan

AJ Arnold, Matthew Berthoud, Justin Cresent, Ada Rigsby
`{ajarnold,mwberthoud,jacresent,acrigsby}@wm.edu`

1 Research Questions

RQ1: *Do apps use all of the permissions they request?*

RQ2: *Are any of the following dangerous permission combinations used in the apps?*

`RECORD_AUDIO & INTERNET` (eavesdropping),

`ACCESS_FINE_LOCATION & RECEIVE_BOOT_COMPLETED` (tracking),

`CAMERA & INTERNET` (stalking),

`SEND_SMS & WRITE_SMS` (use phone as spam bot)

RQ3: *Do apps verify certifications properly? Do they override trust manager or error handler methods so that they accept all certificates?*

RQ4: *Do apps implement improper hostname verification? Do they implement classes or methods that allow all hostnames to be accepted?*

RQ5: *Are apps using deprecated or vulnerable SSL protocols?*

RQ6: *Do apps implement mixed SSL use, making them vulnerable to SSL Stripping attacks?*

RQ7: *Are sensitive data or mutable objects used in any implicit intents?*

RQ8: *Is trusted content loaded within any webviews? If displaying user-provided content, is data loaded into webviews sanitized?*

RQ9: *Are applications that use webviews and `addJavascriptInterface()` correctly using the `@JavascriptInterface` annotation?*

2 Hypotheses

In order to detect instances of permission misuse, SSL API misuse, and interface vulnerabilities, we will use static code analysis through scripting to identify these common issues and oversights that put apps at risk. Based on class lectures and readings, we expect to find these types of security vulnerabilities and malicious behaviors in a significant number of apps. We also expect to find false positives due to the nature of our experiments.

3 Evaluation Plan

3.1 Permissions Misuse Experiment

This experiment tests whether apps utilize all of the permissions they request (**RQ1**), whether different dangerous permission combinations are implemented (**RQ2**), and whether users are prompted and explicitly give their consent to every permission used.

3.1.1 Experimental Setup

We will scrape the MediaStore documentation for what permissions are associated with what constants (eg. `CAMERA` permission with `ACTION_IMAGE_CAPTURE`). After populating this dictionary, we will check to make sure that for every permission there is a corresponding usage of a related API call found in the code, indicating that the permission has been used with MediaStore.

We will use simple string searches in the Manifest to look for the dangerous permission combinations in the manifest.

We will check for lines with `.requestPermissions` and then the name of each permission, to verify that the correct requests are being made for each permission.

3.1.2 Expected Results

Criteria for success include: a MediaStore API call exists for all permissions requested, no dangerous combinations of permissions are found, and a requestPermissions call exists for every permission used within the application.

False positives would be encountered if a dangerous combination is used, but there is no misuse of data, and if both/all permissions in the combination are necessary to the app's core functionality. They would also be encountered if non-MediaStore APIs are used, with proper permissions, since we're only looking at MediaStore APIs, a small subset of all permission-using functions.

3.2 Trust Managers and Error Handlers Experiment

This experiment is to test whether an app overrides a built-in trust manager or error handler to forgo either method's intended purpose of correctly verifying certificates. (**RQ3**)

3.2.1 Experimental Setup

We will do simple string searches by parsing class files to see if the trust manager or error handler methods are overridden.

If so, we'll do some string matching to make sense of what is contained within the newly defined method.

3.2.2 Expected Results

The experiment will succeed if the app contains no overridden trust manager or error handler methods when using SSL APIs, or if the application's overridden trust manager or error handler correctly handles certificates.

We will see some false positives if our string matching doesn't cover all cases. For example, one of these handlers may be overridden in a safe way, with proper handling done in the custom code. Since we're limited by time and scale considerations to static analysis, we may not string-match that code as safe, and it will be marked as a (false) positive.

3.3 AllowAllHostnameVerifier Experiment

This experiment tests whether an app implements the AllowAllHostnameVerifier class, which allows all hostnames to be accepted for a certificate. (**RQ4**)

3.3.1 Experimental Setup

For this experiment, we will parse class files and perform a string search for "AllowAllHostnameVerifier" to see if this class is implemented.

3.3.2 Expected Results

The experiment will succeed if the string is not found and the app does not make use of the AllowAllHostnameVerifier class.

There may be false positives if this string is included in a section of the code that is not actually used for anything (dead code).

3.4 Mixed use SSL Experiment

This experiment tests whether an app contains mixed SSL use, using both HTTP and HTTPS. (**RQ5**, **RQ6**)

3.4.1 Experimental Setup

For this test, we will parse through the class files to see instances of using libraries such as "HttpURLConnection" and check to make sure all are configured to use https consistently. This will involve conducting string searches for "http://", as well as "https://" to see whether the app is only using HTTP or if it is truly mixed use.

3.4.2 Expected Results

The experiment will succeed if the app is found to consistently (all cases) use SSL when it communicates with http. In other words, it will only use https and libraries such as "HttpURLConnection" and the string "http://" will not be found.

There may be false positives if an app makes an http request (not https) to a local server within a secure and controlled environment. While best practice is to use https, due to the nature of where this request was made, we can say that it is not a violation, and thus a false positive.

3.5 addJavascriptInterface Experiment

This experiment tests whether or not the application is being exposed to potentially dangerous outside sources if the application is using a WebView. This interface can create a bridge between a malicious actor and an operating system where the actor is able to inject JavaScript into the WebView. (**RQ7**, **RQ8**, **RQ9**)

3.5.1 Experimental Setup

Using keyword searches, source code will be parsed for the use of addJavascriptInterface(), and if successfully found, the use of the @JavascriptInterface annotation.

3.5.2 Expected Results

The experiment will succeed if the app contains addJavascriptInterface as well as the @JavascriptInterface annotation. This confirms that the developer is only using the interface where necessary. If the app does not contain addJavascriptInterface, this also counts as a success.

There will be false positives if the app includes addJavascriptInterface without any @JavascriptInterface annotations, however it is being used in a safe manner.

References

[1] S. Fahl, M. Harbach, T. Muders, L. Baumgärtner, B. Freisleben, and M. Smith. 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, page

50–61, New York, NY, USA, 2012. Association for Computing Machinery.

[2] F. Shen. Android security via static program analysis. In *Proceedings of the 2017 Workshop on MobiSys 2017 Ph.D. Forum*, Ph.D. Forum ’17, page 19–20, New York, NY, USA, 2017. Association for Computing Machinery.

[1] [2]