Simple Scripts Evaluate Android Application Security

Patrick Collins
ptcollins@email.wm.edu

Abstract

A set of experiments audit a sampling of android applications for security vulnerabilities; focusing on permission misuse, SSL API misuse, and interface vulnerabilities. Relatively simple designs (bash scripts leveraging standard command-line utilities but no external libraries) are able to find numerous security flaws, and what issues there are with the experimental scripts would be somewhat easy to overcome with more resources.

1 Introduction

Many mobile applications available today are insecure. Security is under-prioritized, and rushes to release can impact security as well. [1]

This paper outlines a set of experiments performed to test a sampling of android applications for potential vulnerabilities.

Automated testing like this has the potential to improve the security of mobile applications on the marketplace today, both by raising awareness and thus priority, and by helping developers identify security flaws in their programs.

The remainder of this paper proceeds as follows. Section 2 introduces the specific research questions and experiments. Section 3 describes the design of the experiments. Section 4 evaluates the results of the experiments. Section 5 discusses limitations and potential followups. Section 6 concludes.

2 Overview

The research questions pursued were split between three main categories: Permission Misuse, SSL API Misuse, and Interface Vulnerabilities.

2.1 Permission Misuse

2.1.1 Research Question

- Do apps request combinations of permissions that may potentially be used in a harmful manner?
  - While any permission trades some of a user's privacy away for extra functionality; some combinations of permissions give up more privacy than the sum of the individual permissions.

2.1.2 Hypotheses

- Some apps use both Location and Internet permissions.
  - An application with both of these permissions could track and record your location.
- Some apps use both Microphone and Internet permissions.
  - An application with both of these permissions could eavesdrop on your conversations, recording and uploading their content.
- Some apps use both Camera and Internet permissions.
  - an application with both of these permissions could surreptitiously take and upload pictures and videos.

2.1.3 Experimental Setup

Search through AndroidManifest files to determine whether applications have requested dangerous combinations of permissions.

2.1.4 Expected Results

Successful if it finds all applications listing provided combinations of permissions.

False positive if it finds applications that do not list the provided combinations of permissions.

2.2 SSL API Misuse

2.2.1 Research Question

- Do apps verify certificates incorrectly?

  - Some applications don't take regular precautions to ensure the security of your data and the privacy of your activities; leaving your data open to interception.

2.2.2 Hypotheses

- Some apps trust all certificates.
  - Often through a combination of inexperience and a desire for convenience; developers may set their applications to trust all certificates.
- Some apps ignore SSL errors.
  - Rather than writing extra code to handle SSL errors, sometimes developers simply ignore them.

2.2.3 Experimental Setup

Script-based static analysis of smali code, checking for unimplemented validation functions, as well as certain worrisome strings that indicate permissive security policies.

2.2.4 Expected Results

Successful if it finds all vulnerabilities of the types that it searches for.

False positive if it reports secure or dead code as a vulnerability.

2.3 Interface Vulnerabilities

2.3.1 Research Questions

- Do apps use implicit intents for internal communication?
  - Implicit intents are not guaranteed to reach the intended activity, but inexperience and habit results in their overuse.
- Do apps use broadcasts for internal communication?
  - Broadcasts expose information to external applications unnecessarily when used for internal communication.

2.3.2 Hypotheses

- Some apps call startActivity with intents that they themselves have registered intent filters for.
  - This is can indicate that the startActivity, being called without an explicit activity, is actually meant to trigger another activity in the same application.
- Some apps broadcast intents that they themselves have registered broadcast receivers for.
  - This can indicate that the broadcast is actually meant for internal communication between different parts of the same application; not the intended use of a broadcast.

2.3.3 Experimental Setup

Compare intent actions registered in the AndroidManifest files to string constants used elsewhere within the same application; checking whether or not the methods where those string constants appear also invoke startActivity or sendBroadcast.

2.3.4 Expected Results

Successful if it finds all exported components that are called internally; and identifies where and how they are called from the application.

False positive if it reports components that are also meant for external use.

3 Methodology

This research was accomplished using bash shell scripts, a small bit of awk, and command line utilities like grep, sed, echo, find, sort, and xargs. The scripts were written from scratch, and do not leverage any existing libraries. They were created on Ubuntu 16.04; specifically a patricol/ubuntu_x11 container.

3.1 Permission Misuse

In order to detect when applications have requested permissions in the various dangerous combinations, the script looks at each application's AndroidManifest file. Using simple grep searches, it determines whether or not each permission string (<uses-permission android:name="XYZ"/>) is present in the file.

If any of the combinations being searched for are detected, a line is added to the output to indicate their presence. The output is also sorted by (subjective) level of importance; with microphone access followed by fine location, then coarse location and camera.

3.2 SSL API Misuse

In order to detect when applications have abused the SSL APIs, the script looks through the decompiled smali files for each application.

To improve runtime speed and reduce the number of false positives, a list of assumed-secure directories is

maintained. The list consists of folders used by popular libraries maintained by major companies like Google and the Apache Software Foundation. smali files in those assumed-secure directories are passed over when the script is run.

Each smali file's contents is searched for the presence of a set of worrying strings; whose inclusion generally indicate the employment of an insecure practice.

- SslErrorHandler;->proceed()V: The program proceeds with an operation that was interrupted when an invalid SSL certificate was identified.
- AllowAllHostnameVerifier: The program doesn't verify hostnames.
- ALLOW_ALL_HOSTNAME_VERIFIER: The program doesn't verify hostnames.
- trustAllCerts: The program doesn't check the validity of a certificate.
- trustAllHosts: The program doesn't verify hostnames.

The decompiled smali code is also checked for functions with names that match common security-verifying subroutines.

- checkClientTrusted: Checks the validity of a client's certificate.
- checkServerTrusted: Checks the validity of a server's certificate.
- verify: Verifies an SSL session.

Those functions are then inspected to check whether they are actually implemented; by determining whether or not those functions contain code used to invoke other methods. If a function with a security-verification-based name is detected that appears to be unimplemented as such, it is marked as a potential vulnerability.

3.3 Interface Vulnerabilities

In order to detect when applications have improperly used intents and broadcasts for their own internal communication, a similar approach is employed as was used when detecting SSL API misuse. The smali files are checked for methods containing both the string of an intent action that the application is registered to receive in its AndroidManifest as well as either a startActivity or sendBroadcast call.

Unlike with SSL API misuse, in this case the script needs to check the application's AndroidManifest file to build a list of all of the intent actions that it accepts. Grep is used to find each line, based on the "<action android:name=" signature. Grep's output is then split using xargs and sent to sort, which is also told to keep only unique lines; eliminating duplicates. The lines are then trimmed using command substitution and index slicing.

Those trimmed lines are checked for empty values as well as intent action signatures matching "android.*"; matches of either type are excluded from the final list of registered intent actions. Intent action signatures matching "android.*" are excluded in order to seriously reduce the occurrence of false positives; by far the most common detections were for the ...VIEW action, which applications were often both registered to receive (for types they can handle) and send (for types they cannot; very commonly URIs for the Google Play Store.)

3.4 Detailed Output

When worrisome strings, unimplemented verification functions, or potential interface misuses are detected, the script pulls the entire method declaration containing the potential vulnerability from the smali file; adding it to the output. If the method declaration is very long, the output is appended with only the first line and any lines relatively nearby the line that is the primary cause for concern. That primary detected line is marked as such in the output.

Additional information is also provided at the top of the method declaration; such as the name of the containing file, the line numbers of the start of the method and detected issue, and any locations in other smali files where the containing method is invoked.

Those invocations are collected by searching all the other smali files in the application for lines where a matching method signature is invoked; as well as lines where the method's containing class is initialized.

3.5 Helper Scripts

In addition to the main analyze.sh script that is run on each application; a few extra scripts were created for the convenience of the user.

3.5.1 run.sh

Creates copies of analyze.sh for each application, using xargs to run them in parallel across multiple CPU cores. Once all analyze.sh scripts finish, it removes the copies of analyze.sh and moves and renames the output files for convenience.

3.5.2 clear-output.sh

Removes all created output files; as well as copies of analyze.sh.

3.5.3 read-outputs.sh

Reads all present output files to stdout; regardless of whether or not the program is still running.

4 Evaluation

4.1 Results

4.1.1 Permission Misuse

For the permission misuse tests, the results are as follows:

- 26 Manifests Use Internet & Fine Location.
- 23 Manifests Use Internet & Microphone.
- 22 Manifests Use Internet & Coarse Location.
- 12 Manifests Use Internet & Camera.

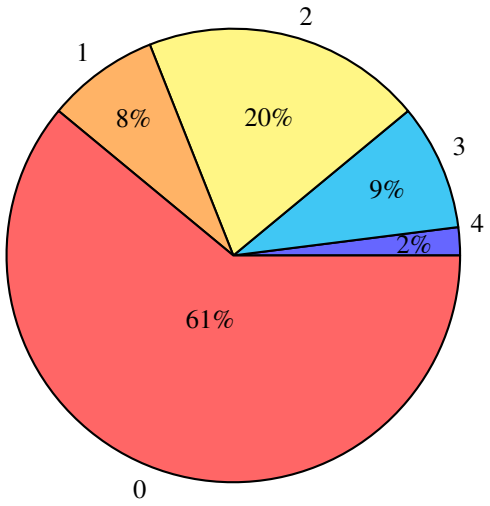

Figure 1: Number of permission combos per application

4.1.2 SSL API Misuse

For the SSL API misuse tests, the results are as follows:

- 43 trustAll Strings Found.
- 27 ALLOW_ALL_HOSTNAME_VERIFIER Strings Found.
- 25 SslErrorHandler;->proceed()V Strings Found.
- 18 AllowAll Strings Found.
- 47 Unimplemented checkClientTrusted Functions Found.
- 43 Unimplemented checkServerTrusted Functions Found.
- 20 Unimplemented verify Functions Found.

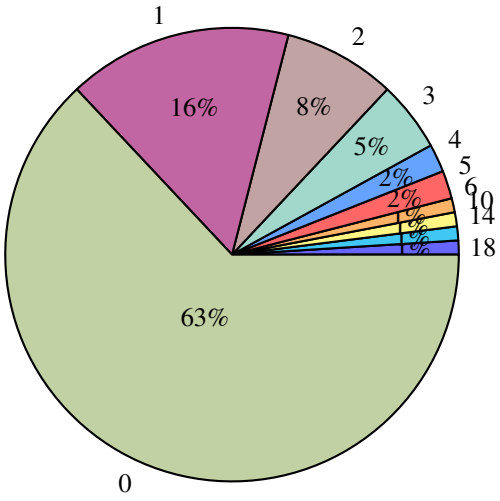

Figure 2: Number of SSL API misuses per application

4.1.3 Interface Vulnerabilities

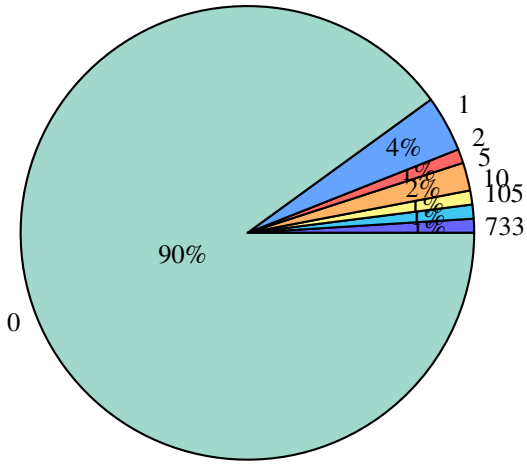

Figure 3: Number of Interface Vulnerabilities per application

4.2 Findings

4.2.1 Permission Misuse

Many applications request dangerous combinations of permissions, and those that do tend to request multiple

dangerous combinations; rather than just one.

No false positives, based on the scope of the experiment.

4.2.2 SSL API Misuse

A large portion of applications had at least one vulnerability; and, of those that had at least one vulnerability, the mean number of vulnerabilities was 3.

Of the small random sampling manually verified, no false positives were found.

4.2.3 Interface Vulnerabilities

Few applications had detected potential interface vulnerabilities, but those that do tend to have many. Of those with at least one potential vulnerability, the mean number of vulnerabilities was 76.

A considerable number of false positives were found in random sampling that were manually verified.

5 Discussion

5.1 Limitations

5.1.1 Permission Misuse

The overall approach is prone to false positives; many applications will justifiably request these combinations of permissions.

The implementation has some potential issues; permissions are expected to be in a standard format (<uses-permission android:name="XYZ"/>) in the file, when in fact a variety of formats (broken into multiple lines, extra spacing, etc.) would be valid.

5.1.2 SSL API Misuse

The lists of worrisome strings and potentially unimplementable functions are extremely incomplete, and this approach could never find every issue on its own; not without triggering an extremely large number of false positives. Checking for known bad implementations is a limited, if simple, approach.

Skipping assumed-secure paths saves time, but still risks missing vulnerabilities.

The method of determining whether a function is unimplemented is not guaranteed to be free of false negatives or false positives.

String constants could be obfuscated, and would be missed by this implementation.

The current approach cannot always be certain that vulnerable code will ever actually be executed.

5.1.3 Interface Vulnerabilities

This approach is very imprecise; causing many false positives. The overall strategy (of checking whether an application sends broadcasts/intents that could but are not guaranteed to be received by itself) is sound, but the implementation is overly simplistic.

Skipping 'android.*' intent actions could cause vulnerabilities to be overlooked.

As with SSL misuse, string constants could be obfuscated, and would be missed by this implementation, and the current approach cannot always be certain that vulnerable code will ever actually be executed.

5.2 Future Possibilities

5.2.1 Permission Misuse

A more in depth approach would be needed; one accompanied by an extra set of assumptions. Potential research could attempt to determine whether the communicated purpose of an application matched with its requested permissions, not only individually; but also as a combination when appropriate. For example, an application that only asks for your location and internet connection should likely mention "tracking" somewhere in its description.

5.2.2 SSL API Misuse

A future implementation could build models of what correct SSL usage looks like, and detect the lack of a correct implementation, rather than relying on recognizing known incorrect implementations.

5.2.3 Interface Vulnerabilities

A more functional design could concretely determine whether any intent broadcast or sent via startActivity is fully compatible with a filter in the same application; by more actively tracking the flow of execution and replicating the intent-acceptance logic.

6 Conclusion

The path to a marketplace full of secure mobile applications is sure to be a long one. Even naively written scripts can find vulnerabilities in many applications, yet those vulnerabilities persist.

Proper accountability and prioritization of mobile application security might be a long way off, but there exists great potential for developing automated systems to test and audit applications - rooting out many flaws.

References

[1] P. I. LLC. The state of mobile application insecurity.