A Formal and Automated Approach for Detecting Embedded Network System Implementation Security Flaws

Mariem Ben Abdallah-Graa [1,2], Nora Cuppens-Boulahia [1], Frédéric Cuppens[1], and Ana Cavalli [2]

1: Telecom-Bretagne, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
2: Telecom-SudParis, 9 Rue Charles Fourier, 91000 Evry - France
Date of starting the thesis: 2010
Co-tutelle: Non
mariem.benabdallah@telecom-bretagne.eu

1 Introduction

Recent years have witnessed an increase in the use of embedded systems such as smartphones. According to a recent Gartner report [6], 417 million of worldwide mobile phones were sold in the third quarter of 2010, which corresponds to 35 percent increase from the third quarter of 2009. To make mobile phones more fun and useful, users usually download third-party applications. For example, we can show an increase in third-party apps of Android Market from about 15,000 third-party apps in November 2009 to about 150,000 in November 2010. These applications are used to capture, store, manipulate, and access to data of a sensitive nature in mobile phone. An attacker can launch flow control attacks to compromise the confidentiality and integrity of the Android system and can leak private information without user authorization. In the study presented in Black Hat conference, Daswani [21] analyzed the live behavior of 10,000 Android applications and show that more than 800 were found to be leaking personal data to an unauthorized server. Therefore, there is a need to provide adequate security mechanisms to control the manipulation of private data by third-party apps. The dynamic taint analysis mechanism is used to protect sensitive data in the Android system against attacks [7]. But this technique does not detect control flows which can cause an under-tainting problem i.e. that some values should be marked as tainted, but are not. Let us consider the attack shown in Figure 1 that presents an under-tainting problem which can be the cause of a failure to detect a leak of sensitive information. The variables x and y are both initialized to false. On Line 4, the attacker tests the user's input for a specific value. Let us assume that the attacker was lucky and the test was positive. In this case, Line 5 is executed, setting x to true and x is tainted. Variable y keeps its false value, since the assignment on Line 7 is not executed and y is not tainted because dynamic tainting occurs only along the branch that is actually executed. As y is not tainted, it is leaked to the network (Line 8) without being detected. Since y has not been modified, it informs the attacker about the value of the

```
x= false;
y=false;
char c[256];
if( gets(c) != user_contact )
    x=true;
else
    y=true;
NetworkTransfer (y);
```

Fig. 1. Attack using indirect control dependency

secret user input. Thus, malicious applications can bypass the Android system and get privacy sensitive information through control flows. This proposed work is on security of embedded systems by detecting the flow control attacks using static and dynamic taint analysis. We propose an enhancement of dynamic taint analysis that propagates taint along control dependencies to track implicit flows in embedded systems such as the Google Android operating system.

2 Related work

Many works exist in the literature to track information flows. They are based on data tainting and static and dynamic analyses for detection of vulnerabilities. Data tainting is implemented in interpreters [20],[11] to analyze sensitive data.

Static taint analysis has been used to detect bugs in C programs [8], [23]. Shankar et al [18] add a new qualifier, tainted, to tag data that originated from an untrustworthy source. JFlow compiler [14] statically checks programs for correctness using information flow annotations and formal rules to prevent information leaks through storage channels. The major disadvantage of all the static analysis approaches is that they require a source code, they have some limitations due to undecidability problems [13] and they might report a number of false positives [3].

A dynamic taint analysis [16],[2],[17],[22] is used at binary level by instrumenting the code to trace and maintain information about the propagation. Thus, this mechanism suffer from significant performance overhead that does not encourage their use in real-time applications.

Privacy issues on smartphones are a growing concern. TaintDroid [7] implements Dynamic taint analysis in real-time applications. Its design was inspired by these prior works, but addresses different challenges specific to mobile phones like the resource limitations. AppFence [10] extends Taintdroid to implement enforcement policies. A significant limitation of these approaches is that they track only explicit flows and they cannot detect control flows. This can cause an under-tainting problem. Some works have been undertaken to solve this under-tainting problem. BitBlaze [19] presents a novel fusion of static and dynamic taint analysis techniques to track implicit and explicit flow. DTA++ [12], based on the Bitblaze approach, presents an enhancement of dynamic taint analysis to limit the under-tainting problem. However DTA++ is evaluated only on benign applications but malicious programs in which an adversary uses implicit flows to circumvent analysis are out of scope. Furthermore, DTA++ is not implemented in embedded application. Trishul [15] correctly identifies implicit flow of information to detect a leak of sensitive information. Fenton [9] proposed a Data Mark Machine, an abstract model, to handle control flows. The Data Mark Machine is based on a runtime mechanism that does not take into account the implicit flow when the branch is not executed. This can cause an under-tainting problem. To solve this problem, Denning [5] inserts updating instructions whether the branch is taken or not to reflect the information flow. Denning and Denning [4] gave an informal argument for the soundness of their compile time mechanism. We draw our inspiration from the Denning approach, but we perform the required class update when Java methods are invoked, as we track Java applications, instead of performing the update at compile time.

These approaches are not implemented in embedded systems like smartphones. Thus, to secure a running process of a smartphone, we propose to prevent the execution of the malicious code by monitoring transfers of control in a program. Then we show that this approach is effective to detect control flow attacks and solve the under-tainting problem.

3 Approach

TaintDroid cannot detect control flows because it only uses dynamic taint analysis. We aim to enhance the TaintDroid approach by tracking control flow in the Android system to solve the under-tainting problem. Trishul is an information flow control system. It is implemented in a Java virtual machine to secure execution of Java applications by tracking data flow within the environment. It does not require a change to the operating system kernel because it analyzes the bytecode of an application being executed. Trishul is based on the hybrid approach to correctly handle implicit flows using the compiled program rather than the source code at load-time. To solve the under-tainting problem in the Android system [1] we use a hybrid approach that improves the functionality of TaintDroid by integrating the concepts introduced by Trishul.

TaintDroid is composed of four modules: (1) Explicit flow module that tracks variable at the virtual machine level, (2) IPC Binder module that tracks messages between applications, (3) File module that tracks files at the storage level and (4) Taint propagation module that is implemented in the native methods level.

To track implicit flow, we propose to add an implicit flow module in the Dalvik VM bytecode verifier which checks instructions of methods at load time. We define two additional rules that we have proved to propagate taint in the control flow. At class load time, we build an array of variables that are modified to handle the branch that is not executed. Figure 2 presents the modified architecture to handle implicit flow.

- Static analysis at load time:
  - We create the control flow graphs which will be analyzed to determine branches in the method control flow. In a control flow graph, each node in

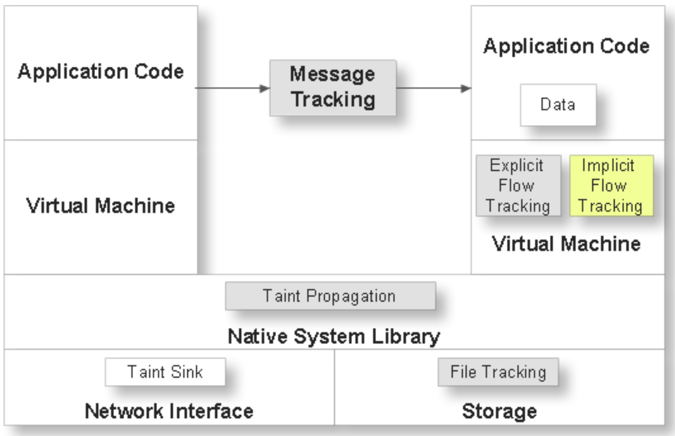

Fig. 2. Modified architecture to handle implicit flow.

the graph represents a basic block. Directed edges are used to represent jumps in the control flow.
- We detect the flow of the condition-dependencies from blocks in the graph.
- We detect variables that are modified in a basic block of the control flow graph to handle not executed branches.

– Dynamic analysis, at run time, uses information provided by the static analysis:
  - We create an array of context taints that includes all condition taints.
  - By referencing to the condition-dependencies from block in the graph, we set the context taint of each basic block.
  - We taint modified variables that exist in the conditional instruction according to the rules of taint propagation: If the branch is taken: $Taint(x) = ContextTaint \oplus Taint(explicitflowstatement)$. If the branch is not taken: $Taint(x) = ContextTaint \oplus Taint(x)$
  - We attach the policies that prevent the use of tainted data in defined taint sink.

4 Results

The implementation of our approach "static analysis at the load time" to handle implicit flows is finished. We have succesfully created the Control Flow Graph from the java application methods. We have formally specified the under-tainting problem. As a solution, we have provided an algorithm based on a set of formally defined rules that describe the taint propagation. We have proved the completeness of those rules and the correctness and completeness of the algorithm. We have proved that our system cannot create under tainting states.

5 Conclusion and future work

In order to protect embedded systems from software vulnerabilities, it is necessary to have automatic attack detection mechanisms. In this work, we show how to enhance dynamic taint analysis with static analysis to track implicit flows in the Google Android operating system. We prove that our system cannot create under tainting states. Thus, malicious applications cannot bypass the Android system and get privacy sensitive information through control flows. Future work will be to test our approach and show that it resists to code obfuscation. Once the implementation is finished, we will be able to evaluate our approach in terms of overhead and false alarms.

References

1. Android, `http://www.android.com/`
2. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: ISCC'06. Proceedings. 11th IEEE Symposium on. pp. 749–754. IEEE (2006)
3. Chess, B., McGraw, G.: Static analysis for security. Security & Privacy, IEEE 2(6), 76–79 (2004)
4. Denning, D., Denning, P.: Certification of programs for secure information flow. Communications of the ACM 20(7), 504–513 (1977)
5. Denning, D.: Secure information flow in computer systems. Ph.D. thesis, Purdue University (1975)
6. Egham, U.: Gartner says worldwide mobile phone sales grew 35 percent in third quarter 2010; smartphone sales increased 96 percent (November 2010), `http://www.gartner.com/newsroom/id/1466313`
7. Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., Sheth, A.: Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX conference on Operating systems design and implementation. pp. 1–6. USENIX Association (2010)
8. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. Software, IEEE 19(1), 42–51 (2002)
9. Fenton, J.: Memoryless subsystem. Computer Journal 17(2), 143–147 (1974)
10. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 639–652. ACM (2011)
11. Hunt, A., Thomas, D.: Programming ruby: The pragmatic programmer's guide. New York: Addison-Wesley Professional. 2 (2000)
12. Kang, M., McCamant, S., Poosankam, P., Song, D.: Dta++: Dynamic taint analysis with targeted control-flow propagation. In: Proc. of the 18th Annual Network and Distributed System Security Symp. San Diego, CA (2011)
13. Landi, W.: Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS) 1(4), 323–337 (1992)
14. Myers, A.: Jflow: Practical mostly-static information flow control. In: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages. pp. 228–241. ACM (1999)

15. Nair, S., Simpson, P., Crispo, B., Tanenbaum, A.: A virtual machine based information flow control system for policy enforcement. Electronic Notes in Theoretical Computer Science 197(1), 3–16 (2008)
16. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer (2005)
17. Qin, F., Wang, C., Li, Z., Kim, H., Zhou, Y., Wu, Y.: Lift: A low-overhead practical information flow tracking system for detecting security attacks. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture. pp. 135–148. IEEE Computer Society (2006)
18. Shankar, U., Talwar, K., Foster, J., Wagner, D.: Detecting format string vulnerabilities with type qaualifiers. In: Proceedings of the 10th conference on USENIX Security Symposium-Volume 10. pp. 16–16. USENIX Association (2001)
19. Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M., Liang, Z., Newsome, J., Poosankam, P., Saxena, P.: Bitblaze: A new approach to computer security via binary analysis. Information Systems Security pp. 1–25 (2008)
20. Wall, L., Christiansen, T., Orwant, J.: Programming perl. O'Reilly Media (2000)
21. Wilson, T.: Many android apps leaking private information (July 2011), `http://www.informationweek.com/security/mobile/many-android-apps-leaking-private-inform/231002162`
22. Yin, H., Song, D., Egele, M., Kruegel, C., Kirda, E.: Panorama: capturing system-wide information flow for malware detection and analysis. In: Proceedings of the 14th ACM Conference on Computer and Communications Security. pp. 116–127. ACM (2007)
23. Zhang, X., Edwards, A., Jaeger, T.: Using cqual for static analysis of authorization hook placement. In: Proceedings of the 11th USENIX Security Symposium. pp. 33–48 (2002)