Raindroid – A System for Run-time Mitigation of Android Intent Vulnerabilities

[Poster]

Bradley Schmerl* Jeffrey Gennari† Javier Cámara* David Garlan*

*{schmerrl,jcmoreno,garlan}@cs.cmu.edu
Institute for Software Research
Carnegie Mellon University
Pittsburgh, PA 15213

†jsg@sei.cmu.edu
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

ABSTRACT

Modern frameworks are required to be extendable as well as secure. However, these two qualities are often at odds. In this poster we describe an approach that uses a combination of static analysis and run-time management, based on software architecture models, that can improve security while maintaining framework extendability. We implement a prototype of the approach for the Android platform. Static analysis identifies the architecture and communication patterns among the collection of apps on an Android device and which communications might be vulnerable to attack. Run-time mechanisms monitor these potentially vulnerable communication patterns, and adapt the system to either deny them, request explicit approval from the user, or allow them.

Categories and Subject Descriptors

D.2.11 [**Software Architectures**]: domain-specific architectures; D.4.6 [**Security and Protection**]: Information flow controls, invasive software

General Terms

Security, Design

Keywords

software architecture, security, self-adaptation

1. INTRODUCTION

Software frameworks are used ubiquitously in modern applications, in the commercial sector and increasingly in the defense sector as well, because they offer a unique means for achieving composition and reuse at scale. Achieving security in framework-based applications, however, can be challenging because of the close coupling between a framework and its plugins: Plugin developers must understand and obey the constraints in the framework's security model, which can be quite complex, in order to achieve security of the resulting application.

There is an inherent trade-off between constraining what plugins can do within a framework to achieve goals such as security, and being open to allow a more diverse ecosystem. The mobile device arena is an interesting case in point. The Android framework provides flexibility of communication between apps (plugins that use the Android framework) that allows other apps to provide alternative core functionality (such as browsing, SMS, or email) or to tailor other parts of the user experience. However, this flexibility also means malicious apps can be written that take advantage of this flexibility.

Android uses an event style that allows flexible communication between apps, where events are called *intents*. An intent can be explicitly sent to a known app, or implicitly sent to any app that may be interested in it. While intents provide a great deal of flexibility, they are also the source of a number of security vulnerabilities such as intent spoofing, privilege escalation, and unauthorized intent receipt [5]. To some degree, these vulnerabilities can be uncovered by analyzing apps and performing static analysis to see how intents are used, what checks are made on senders and receivers of intents, and so on [13]. However, Android is an extendable platform that allows users to dynamically download, update, and delete apps that makes a full static analysis impossible. Furthermore, static analysis can detect vulnerabilities, but not actual exploits (which happen at run time), meaning that false positives could lead to lower flexibility of the Android device, for example by restricting all communication between apps to mitigate the vulnerability. Therefore, a combination of static analysis and dynamic adaptation could help to provide the benefits of security while still allowing for such flexibility and extendability.

In this poster, we illustrate an approach that combines information from static analysis with run-time monitoring and mitigation. In a previous poster [14] we described the definition of an architecture style for analyzing security properties in Android. In this poster, we build on this work and describe a prototype for Raindroid, a run-time adaptation service for Android that leverages previous work on architecture-based self-adaptation and static analysis of Android security vulnerabilities.

HotSoS'16 Pittsburh, PA, USA

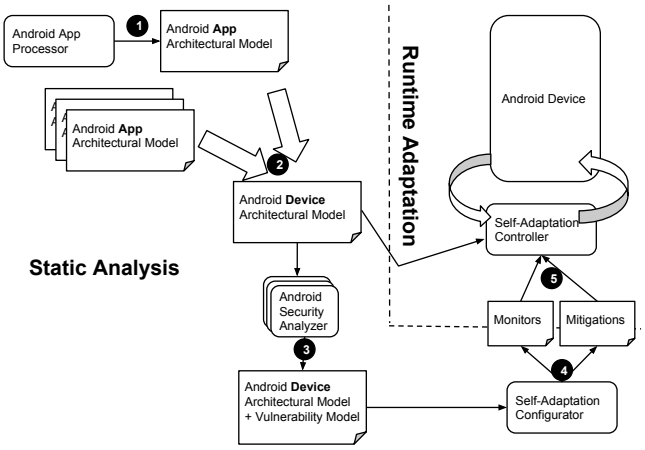

Figure 1: Combining Static Security Analysis with Runtime Adaptation

2. APPROACH

Our approach comprises both static analysis elements and run-time adaptation elements, and is depicted in Figure 1. An Android App Package (containing the manifest[1] for the application, and the byte code) is processed to produce an architectural model (❶). This can be done in a variety of ways, including those described in [1] or [13]. The result is an architectural model of the app, which specifies the run-time configuration of the components comprising the app and their connections.

Security vulnerabilities in Android often occur through the interaction of apps, however, and so this model needs to be combined with models of other apps on the device to produce an Android Device Architectural Model (❷). This model can then be analyzed for security vulnerabilities using a number of techniques, such as architectural data flow analysis [8, 15] and model checking [13] (❸). The output from this analysis is a set of locations and behaviors in the model that are potential vulnerabilities.

This information is passed to a Self-Adaptation Configurator that decides how to monitor the apps, conditions that indicate how to detect that a vulnerability is being exploited, and mitigations to prevent, secure, or advise users about potentially dangerous interaction (❹). These steps, ❶-❹, can be done statically before an app is installed on a device.

The information (as well as the architectural model) is used by a Self- Adaptation Controller (❺) that uses the monitors to detect at run time security problems and decide mitigations to perform. This controller is based on the Rainbow architecture-based self-adaptation framework that we have successfully used in a variety of areas [9, 2].

3. ANDROID ARCHITECTURE STYLE

A key step in this approach is to create a formal architectural model of the Android framework in order to have a basis for modeling the structures and constraints in Android, and to permit analysis of security vulnerabilities and exploits. To do this, we have developed an architectural style in Acme [7], that represents intent interactions and permissions in Android. An example of an architectural model for a simple PhotoStream app is shown in Figure 2.

[1]The manifest describes the classes for the app, the permissions, the allowed communications and permissions

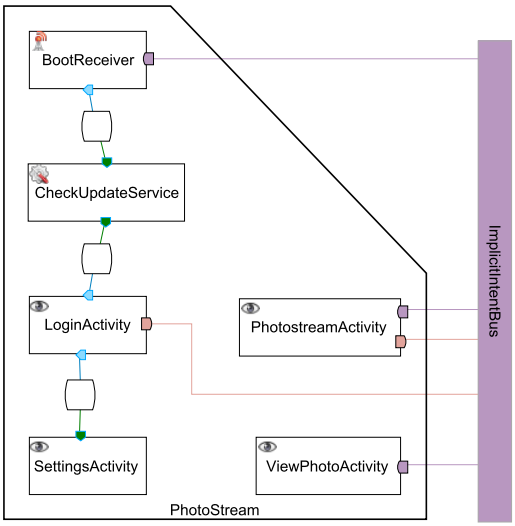

Figure 2: PhotoStream App Model in Acme

The style represents explicit intents as direct connections between Android components, separating them from implicit intents that are communicated through the intent bus.

Components in an app are grouped together so that permission usage within the app can be checked by static analysis (e.g., components of an app can only use the permissions granted to the app) and verified dynamically. Other apps on a device are represented in their own groups, also connected to the intent bus. Modeling apps, permissions, and intents in this way allows us to write constraints that check for privilege escalation and other data flow vulnerabilities in the model. The model can be used to capture information from static analysis of apps, and is also used for run-time monitoring. For example, if static analysis detects a potential vulnerability in the intent to use the photo resources sent by the PhotoStreamActivity in Figure 2, run-time monitors can be placed in that activity to check that photos are sent only to applications that have the appropriate permissions.

4. RAINBOW FOR SELF-ADAPTATION

In prior work, we have shown that architectural models work well for reasoning about (a) whether a run-time system is exhibiting its designed quality attributes and (b) the best adaptations to perform on the system that trade-off among the different quality attributes when those quality attributes are not being met. These notions for run-time adaptation are embodied in a framework called Rainbow [6, 3]. The Rainbow framework uses software architectures and a reusable infrastructure to support self-adaptation of software systems. Figure 3 shows the adaptation control loop of Rainbow. Probes are used to extract parameters from the target system that update the model via Gauges, which abstract and aggregate low-level information to detect architecture- relevant events and properties. The architecture evaluator checks for satisfaction of constraints in the model and triggers adaptation if any violation is found, i.e., an adaptation condition is satisfied. The adaptation manager, on receiving the adaptation trigger, chooses the "best" strategy to execute, and passes it on to the strategy executor, which executes the strategy on the target system via effectors.

The best strategy is chosen on the basis of stakeholder utility preferences and the current state of the system, as

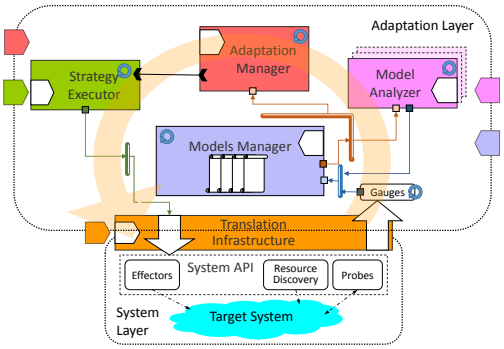

Figure 3: Rainbow Self-Adaptation Framework

reflected in the architecture model. The underlying decision making model is based on decision theory and utility; varying the utility preferences allows the adaptation engineer to affect which strategy is selected. Each strategy, which is written using the Stitch adaptation language [4], is a multi-step pattern of adaptations in which each step evaluates a set of condition-action pairs and executes an action, namely a tactic, on the target system with variable execution time. A tactic defines an action, packaged as a sequence of commands (operators). It specifies conditions of applicability, expected effect and cost-benefit attributes to relate its impact on the quality dimensions. Operators are basic commands provided by the target system.

As a framework, Rainbow can be customized to support self-adaptation for a wide variety of system types. Customization points are indicated by the cut- outs on the side of the architecture layer in Figure 3. Different architectures (and architectural styles), strategies, utilities, operators, and constraints on the system may all be changed to make Rainbow reusable in a variety of situations.

5. RAINDROID

Rainbow provides a bridge between static analysis and run-time monitoring and adaptation of the Android platform. However, there are a number of challenges that need to be addressed so that it can be employed in the Android context. Firstly, we need a method to extract information from Android devices and to affect the behavior of them that Rainbow can integrate with. This method needs to leave the byte code of the application untouched (so that app checksums remain unchanged and so the standard play store can be used). Second, the static analysis required to check for vulnerabilities is not able to run on the Android platform; it uses model checking via Alloy[10] that requires resources beyond most Android platforms. We therefore need to devise a method to connect the Android probes and effectors that we do develop to a remote server that runs the COVERT analysis and Rainbow decision-making. Third, Rainbow was primarily designed as a reactive self-adaptive system that runs independent of the target system. For security purposes, Rainbow needs to be preemptive, quickly detecting potentially bad situations and stopping them if necessary. Finally, because of the need for running the analysis and decision-making on a remote host, we need a method of self-adaptation that works in disconnected mode (i.e., when the Android device is not connected to the network, or the server is otherwise unreachable)[11].

To address these challenges we are in the process of developing Raindroid, a service on Android that connects Android to Rainbow. Raindroid consists of two components:

- *An Android application that provides a service to connect to Rainbow.* Currently, Raindroid is targeted to the scenario of Android intents (as described above). Information gathered from apps in Android is sent via this service to Rainbow. For example, events such as apps starting or sending intents are sent to Raindroid. These events are sent to Rainbow for further analysis. Rainbow effects changes on the Android device by communicating to Raindroid the allowed communication paths (those that have been determined safe), and *postures* that describe how to handle other (suspicious) intents. Postures are coarse-grained policies describing what should be done when particular events occur. We currently have implemented three postures related to intents: prevent the intent from being sent, ask the user of the device to approve, and continue the intent sending as normal. Rainbow decides the best posture based on the set of active apps. For example, if a banking app has been opened, Raindroid will automatically prevent any non-approved intents from being sent.
- *An Xposed module that instruments each Android application.* The module sends events to Raindroid, and affects the behavior of intent sending APIs in Android. Xposed[2] is a framework that runs on rooted Android devices, and modifies the Android kernel to allow third party plugins to weave their own code into the Android application. Xposed modules indicate the Java methods they wish to affect. Currently we intercept the application start-up methods to report when an app is started, and methods that send intent (e.g., *startActivity*). For the latter, we use this to report the intent being sent to the Raindroid Service, and to process the posture that is sent to it from Rainbow.

The challenges of disconnected operation and preemption are addressed through the notion of *postures* that are described above. Raindroid acts as a proxy to Rainbow, making quick decisions and preventing or allowing behavior that Rainbow has previously analyzed. If the Android device becomes disconnected from Rainbow, the posture implementation is still executed because it runs locally on the Android device. Similarly, because Raindroid runs on the Android device and uses Xposed to change method behaviors, it can run synchronously with the Android application to prevent or accept certain behaviors. For example, the way that we change the behavior of the *startActivity* method of Android for sending intents is to add the following steps:

1. Report the intent to Raindroid;
2. Wait for the posture to be sent;
3. Implement the posture to prevent the intent from being sent, open a screen to get approval from the user, or send the intent as originally intended.

The decision about which posture to invoke is done quickly (without connection to Rainbow, but based on prior Rainbow analysis) and so can be done synchronously with the Android app.

[2] `http://repo.xposed.info/`

5.1 Integration with Rainbow

On the Rainbow side, gauges listen for events from Rainbow and process them in the following ways:

Device connection. When a device first connects to Rainbow, Raindroid communicates the list of all installed apps. A gauge then looks up the Acme architecture model for the app and loads this into Rainbow. If the architecture model is not found, it generates the architecture model from the APK for the app using an Acme converter that takes the graph generated by COVERT and translates it into Acme.

App start/stop. The model is updated to indicate whether the app is active or inactive on the device, and is used to help determine the posture that should be sent.

Intent sent/received. This information is used to update the communication paths in the model, and to detect whether vulnerable or unanalyzed paths are being used. This also helps to decide the posture to be sent back to Raindroid.

Once the architecture model of the device is loaded into Rainbow it is then analyzed using COVERT to determine intents that represent a vulnerability. The list of intents that are not determined vulnerable are sent to Rainbow, along with the posture, so that Raindroid has a list of valid intents and knowledge about what to do if an unknown or vulnerable intent is detected.

6. EXAMPLE

We demonstrate Raindroid in a scenario that involves dynamic code loading. Android apps may dynamically load code from another site or file on the device, and then through Java reflection call this new code. Because the code is dynamically loaded, it thwarts any static analysis that might be done. There are a number of reasons that apps use dynamic code loading: for example, if apps use third-party frameworks, these frameworks can provide updates through dynamic code loading independent of the play store; many frameworks use dynamic code loading to download and display advertisements. Work described in [12] shows that Android does not enforce appropriate checks when apps use external code, and that developers often are not aware of the risks and do not add their own checking mechanisms. Thus, this is a significant source of vulnerability in Android.

The Raindroid demonstration involves a custom app that we developed that loads a piece of code dynamically. This code accesses the Contacts on an Android device and emails this information to a third party (the email is sent by sending an intent to the email app). Static analysis of the original app does not indicate that this intent is being sent, and so it is not sent to Raindroid as a member of the allowable set of intents. When Raindroid detects that the intent is being sent (and that it is not known), it (a) sends this information to Rainbow and (b) sends back the appropriate disposition of the intent based on the posture.

Acknowledgments

This work is supported in part by the National Security Agency. The views and conclusions contained herein are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Security Agency or the U.S. government.

7. REFERENCES

[1] M. Abi-Antoun, S. Chandrashekar, R. Vanciu, and A. Giang. Are object graphs extracted using abstract interpretation significantly different from the code? In *SCAM 2014*, 2014.

[2] J. Cámara, P. Correia, R. de Lemos, D. Garlan, P. Gomes, B. Schmerl, and R. Ventura. Evolving an adaptive industrial software system to use architecture-based self-adaptation. In *SEAMS'13*, 2013.

[3] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, May 2008. Institute for Software Research Technical Report CMU-ISR-08-113.

[4] S.-W. Cheng and D. Garlan. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software, Special Issue on State of the Art in Self-Adaptive Systems*, 85(12), December 2012.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *MobiSys '11*, 2011.

[6] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), October 2004.

[7] D. Garlan, R. T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In *Foundations of Component-Based Systems*. 2000.

[8] D. Garlan and B. Schmerl. Architecture-driven modelling and analysis. In *Proceedings of the 11th Australian Workshop on Safety Related Programmable Systems (SCS'06)*, 2006.

[9] D. Garlan, B. Schmerl, and S.-W. Cheng. Software architecture-based self-adaptation. In *Autonomic Computing and Networking*. Springer, 2009.

[10] D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[11] G. A. Lewis and P. Lago. A catalog of architectural tactics for cyber-foraging. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures*, QoSA '15, pages 53–62, New York, NY, USA, 2015. ACM.

[12] S. Poeplau, Y. Fratantonio, A. Bianchi, C. Kruegel, and G. Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, 2014.

[13] A. Sadeghi, H. Bagheri, and S. Malek. Analysis of android inter-app security vulnerabilities using COVERT. In *ICSE 2015*, 2015.

[14] B. Schmerl, J. Gennari, and D. Garlan. An architecture style for android security analysis: Poster. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security*, HotSoS '15, pages 15:1–15:2, New York, NY, USA, 2015. ACM.

[15] R. Vanciu and M. Abi-Antoun. Ownership object graphs with dataflow edges. In *20th Working Conference on Reverse Engineering (WCRE)*, 2013.