Detection and Prevention of Misuse of Software Components

DAI TING

(*B.Eng., TSINGHUA UNIVERSITY*)

A THESIS SUBMITTED

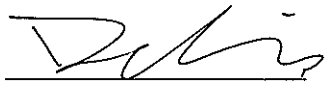
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2015

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



Dai Ting

21 January 2015

Acknowledgements

First, I would like to thank my advisers, Professor Zhenkai Liang and Professor Roland H. C. Yap, for their constant advice and help on my varied research interests in my candidature. They encourage me on both my work and life since the first year of my study. Without their guidance, I would never made any steady progress on my research. Most importantly, they enlighten me with the power of thinking independently, which makes a solid foundation for my career and will benefit me in the future.

Second, I am grateful to all of my collaborators over the years for their support and sharing of experiences. Especially, I would like to thank Sai Sathyanarayan, Mingwei Zhang, Behnaz Hassanshahi and Xiaolei Li. I benefit a lot from working together with them. I would also like to thank all my labmates for their help on my study and life in campus, especially Utsav Saraf, Bodhisatta Barman Roy, Hong Hu, Yaoqi Jia, Guangdong Bai, Xinshu Dong, Kailas Patil, Zheng Leong Chua, Ziqi Yang, Xuhui Liu, Benjamin Thian, Dongyan Zhang, Jiangang Wang, Yue Chen, Yongzheng Wu, Wei Xia, Liming Lu, Jia Xu, Xuejiao Liu, Chengfang Fang, Chunwang Zhang, Xiaolu Zhu, Zhaofeng Chen, Loi Luu, Shweta Shinde, Shruti Tople, Enrico Budianto, Inian Parameshwaran, and Pratik Soni.

Finally, I would like to thank all my family and friends for their help and trust in me. They encourage me a lot and support me with great effort. I am lucky to have them by my side.

Contents

Abstract

Software components are building blocks of modern computer systems. Modern computer systems are complex integrations of software components that encapsulate many sets of software functionalities. Moreover, these functionalities are affected by both the interactions of software components and various system settings. As a result, software components can behave in an unexpected way, deviating from the original functionality provided by the software components. We call such a vulnerability, a *component misuse vulnerability*.

In this thesis, we propose systematic solutions to detect and prevent the component misuse vulnerability. First, binary loading behaviors in software programs can be misused via manipulating various system settings. We develop an approach to detect such binary loading vulnerabilities. It explains binary loading behaviors by listing various system settings, binaries and files that can affect the loading behaviors. It detects the unexpected uses of binary loading by identifying the factors that can be controlled by the attackers. Second, the APIs in ActiveX controls can be misused via component interactions. We develop a mechanism to detect ActiveX API-misuse vulnerabilities in Internet Explorer (IE), by detecting unexpected uses of APIs in the component interactions. It also blocks the APIs that can be misused in a fine-grained manner. The problem of component misuse also manifests itself in other systems. We propose a solution to mitigate the damage caused by component misuse vulnerability due to the interactions of software components in the Android platform. In particular, we develop a prevention mechanism of privilege escalation in Android inter-component communication (ICC). It prevents permission re-delegation in Android ICC which can cause the sender application (app) to gain additional privileges from the recipient app.

List of Tables

List of Figures

Chapter 1

Introduction

Software components are building blocks of software programs. They encapsulate many sets of software functions to support various software features. On one hand, these components facilitate code reuse; on the other hand, they also bring complexity into the computer systems.

Modern computer systems are complex integrations of these software components. The complexity of these systems are further amplified by the interaction of software components and the system settings. The interactions of the software components add various dependencies to the computer systems. These dependencies further complicate computer systems, which can result in problems like *dependency hell*.[1] Moreover, various system settings are created to adjust the functionality of these components. For example, system configurations and environment variables are created to accommodate software components, but they can also cause an insecure library loading attack.

In complex computer systems, the subtle interactions among software components can result in unwanted behaviors or even attacks. For example, the "Microsoft Office Snapshot Viewer" ActiveX control is designed to view and print snapshots of document files in JavaScript. The control is protected in the browser sandbox, and provides an API to specify the path of snapshot files. However, when interacting with the browser components, this API offers implicit functionality to allow JavaScript to write files in the local file system, effectively bypassing the browser sandbox. This can be leveraged by attackers to overwrite files in the local file

[1] The frustration of users who have installed software packages which have dependencies on specific versions of other software packages. [70]

system. The dependencies from software components on the system settings can also result in unwanted behaviors. A subtle example is the following Safari "Carpet Bomb" attack [1]. The Safari browser can download files automatically to the desktop directory and the download directory in Windows without user interactions. Many Windows programs, e.g. Internet Explorer (IE), load binaries according to system settings, such as "binary search order". IE can be leveraged by the attackers to load arbitrary binaries downloaded by Safari, if this binary search order includes the desktop directory. In this example, neither Safari nor IE alone can cause the attacks. It is the subtle interaction among the components, and the various system settings that result in this attack.

Component interactions and system settings of complex systems open up various attack surfaces [87] of software components. When components are interacting with each other, they may be susceptible to the attacks shown in the above examples. In these attacks, the software components are "misused" by attackers in a way they are not designed for. An API is misused in the ActiveX example to allow file overwrite instead of showing snapshots. Binary loading behaviors are misused in the Safari "Carpet Bomb" attack to allow loading of binaries downloaded arbitrarily. We call such behaviors *component misuses* in this thesis.

The misuse of software component is a common problem in complex computer systems. It is not confined to desktop systems as shown in the above examples; the Android platform also has similar problems. Android has a different component interaction model, where its components share functionalities using an intent-based mechanism. However, the mechanism can be misused to allow applications (apps) to escalate their privileges, instead of running with their designated privileges.

Component misuses are caused by limited understanding of the software components by programmers and software designers, which is further amplified by the overall complexity of the computer systems, including complicated dependencies and component interactions. Software components are traditionally designed for functionality and flexibility, which may have inadequate or incomplete consideration of security. It is common that the software functionality is not always clearly specified in software specifications. Even when the default functionality is clear, the interactions among components can further hinder the understanding. Furthermore,

the system settings and software configurations can affect and alter the original functionality of the components. The misused component behaviors have dependencies on these settings.

In this thesis, we aim to detect and prevent the misuse of software components. We first develop solutions for identifying the causes of component misuse, which are the dependencies from component functionalities to system settings and component interactions. Then, we secure the usage of software components by regulating component interactions to prevent component misuses.

The component misuse problem is challenging to solve. First, the misuse arises from the normal functionality of software components. The designers may not be aware of the subtle impacts that the normal functionality can bring. One example is the Android inter-component communication, which is designed to support sharing of functionalities. But it also implicitly results in various forms of privilege escalation [49]. Furthermore, although the components themselves are not malicious, users of these components can misunderstand the original functionality and use them in an unexpected way. This implies traditional malware detection techniques [43, 119] and anti-virus detectors are not applicable on this problem. The misused components have vulnerabilities at a different level compared to the prevalent software vulnerabilities, such as memory-corruption errors, buffer overflows and integer overflows addressed by existing solutions [22, 34, 37, 58, 81, 90]. The vulnerability of component misuse comes from the logic of the design of the functionalities. This requires us to come up with a different solution than the existing ones.

Second, the misuse can be caused by the interactions with the system environments and settings on the software components. This interaction can be viewed as a *dependency* from the unexpected component behaviors on the system environment and settings. For example in the Safari "Carpet Bomb" attack, the system setting "binary search order" affects the loading of software components. As a result, various dependency analyses [23,26,83,96,100,105,109] can be applied to this kind of misuse problems. To be specific, backward dependency analysis [26] can be applied to explain the cause of the misuse in component behaviors. In this way, we can provide better understanding of component functionalities by showing various dependencies, so that the misuse can be avoided.

Third, the misuse can also be caused by the interactions among the software components themselves. A component can function safely by itself. But when it is interacting with other components, unexpected functionalities can be triggered, like the ActiveX example. A complex computer system contains many interactions from components. Although it is difficult to clearly identify the impact of these interactions on the functionality of components, the impact can be modeled as a *path* between the normal entry functionality and the unwanted functionality when a misuse happens, when interactions among software components are modeled as a graph. In the ActiveX example, the API for specifying the snapshot file unintentionally bridges the gap between the browser component and the native API to write files. This forms a path that can be analyzed in a way similar to analyzing attack path [92, 115]. However, instead of infiltrating the system or accessing private information, this path leads to functionalities that are normally unavailable to software components. Through a reachability analysis, we are able to detect the vulnerability of misuse caused by component interactions.

The consequence of component misuse is gaining more functionality than needed. For example, privileges can be gained through interactions in Android inter-component communications. This requires us to design a solution to limit the functionalities. This solution should be effective and simple to deploy. One alternative is to directly block the entries to the unwanted functionality of software components. This effectively cuts off the *path* between the normal functionality and the unexpected behaviors. For example, directly blocking the use of the vulnerable ActiveX APIs cuts off the path between the normal functionality and the file overwrite functionality. This solution should also be practical and efficient. It should limit the functionality of software components and not lose the usability of the components. For example, in Android, because of the specific designed functionality sharing mechanisms, blocking the interactions in a restrictive way [57] can affect the usability of the components when they are interacting with other components. Another example is in the existing ActiveX security mechanisms. Due to the coarse-grained design, blocking certain functionalities usually means losing all the usability of the ActiveX controls. What we need is a practical solution to prevent component misuses and preserve the usability of the software components at the same time.

1.1 Our Solutions

We propose a line of systematic solutions to detect and prevent misuses in software components. First, we inspect the individual software component, whose functionalities are affected by program configurations and system settings. We explain in detail the binary loading behaviors in software components and find out the ones that can be controlled by attackers. Second, with the understanding of individual software components influenced by complex computer systems, we identify how software components affect each other in the component interactions. We detect the misuse of dangerous APIs in ActiveX controls. Finally, we propose a solution to prevent component misuses in the component interactions, and preserve the usability of the components. We apply the solution on both ActiveX controls and Android components. It prevents the dangerous ActiveX APIs from being invoked. It also prevents the privilege escalation from the misuse of interactions in Android components, where the component interactions follow a specified model. More specifically, our solutions are as follows:

Detecting Loading Misuse in Software Components. Loading software component is an essential step in program execution in most operating systems. Due to the complexity of various system settings and software configurations in computer systems, the loading behaviors can be misused, allowing the attackers to misuse the loading behaviors to achieve unintended loading of binaries into the program, such as the loading behavior in the Safari "Carpet Bomb" attack. Successful loading of attacker provided binaries can lead to remote code execution or privilege escalation.

In order to detect such binary loading vulnerabilities, we develop an analysis mechanism using data dependency and system dependency to explain the binary loading behaviors. We extract factors, such as system configuration and software specification and various system settings and paths, into a loading dependency graph (LDG) to model the loading behavior. It describes why and how the loading behaviors happen. With this solution, we provide a better understanding of the interactions among software components and complex system configurations and settings, so that the vulnerable loading of software components can be detected.

Detecting API Misuse in Software Components. As described in the previous ActiveX example, the functionality of software components can be misused to allow the embedding application of software components to access the unwanted functionality. We call such vulnerabilities, *API-misuse vulnerabilities*.

We present a solution to detect API misuse vulnerabilities in ActiveX controls in Microsoft Windows. We construct reachability models to identify which ActiveX APIs can reach system APIs. By applying this solution, we provide an explanation on how component interactions cause misuses, and a prevention mechanism by cutting off the paths between the components. With this part of the work, we reason about how component interactions affect the component behaviors and cause the component misuse.

Preventing Re-delegation Misuse in Software Component. Many software components rely on interacting with other components to extend their functionality. Such behaviors are very common in Android platform. An app can request another suitable app to gain system services. In fact, the suitable app is exercising its own permissions on behalf of the requesting app. This is called permission re-delegation, where the requesting app indirectly obtains additional permissions from the suitable app.

We propose a security framework UPPDROID to mitigate permission re-delegation attacks in Android platform. UPPDROID leverages resource virtualization to allow the accesses to the system resources that are guarded by Android permissions. It prevents the permission re-delegation in Android ICC that can allow the sender to gain additional privileges from the recipient. With the help of resource virtualization, the usability of this prevention mechanism is enhanced.

In summary, we provide the following main contributions in this thesis:

1. We propose a series of solutions towards detecting and preventing component misuse vulnerability caused by various aspects of complex computer systems in Windows and Android platforms.

2. We provide a mechanism to explain how the uses of software components are affected by different settings of complex computer systems. We detect real world binary loading

vulnerabilities in Windows and provide explanations of the causes.

3. We provide a mechanism to explain how software component behaviors are affected by component interactions from complex computer systems. We detect real world ActiveX API misuse vulnerabilities in Internet Explorer in Windows.

4. We prevent the misuses in component interactions and preserve the usability in software components. We apply this idea to prevent privilege escalation in Android platform and API-misuse attacks in ActiveX controls in Windows.

1.2 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 gives background knowledge on Windows and Android platforms. We also show and discuss existing security mechanisms in Windows binaries, ActiveX and Android component interactions. We further discuss existing solutions for vulnerability detection and attack path analysis. We present our LDRSCOPE system on understanding binary loading behaviors in Chapter 3, which shows how system settings affect component behaviors. Chapter 4 presents our work on detecting and preventing API-misuse vulnerabilities, which shows how component interactions affect component behaviors. Chapter 5 presents our prevention mechanism of component misuses in Android platform. Finally, we conclude the thesis and discuss the future work in Chapter 6.

Chapter 2

Background

In this chapter, we introduce the background for the rest of the thesis. We discuss the design of the Windows and Android platforms, where our solutions are applied. We introduce the main security mechanisms on these platforms. We also survey the existing techniques, such as vulnerability detection and attack path analysis. They are related to our solutions in identifying the misuse of components. In general, the component misuse problem addressed in this thesis concerns partly the "intentional, non-malicious flaws", which is caused by functional requirements that are written without regard to security requirements, and partly the "inadvertent, identification/authentication flaws", which allows a protected operation to be invoked without sufficiently checking, based on genesis discussed by Landwehr et al. [79]. In another word, the functionality of the software component may not contain malicious purpose, but, they are used in an unsafe way. The component interactions can cause unexpected behaviors by the designers in the software components.

2.1 Windows Binary and Components

The Windows platform has a number of ways in using software components, especially the binaries. We present the basics of Windows binaries and components, such as ActiveX controls. We also discuss how binary loading works for Windows in this section.

Windows Binaries. Windows binaries are the essential units of programs, forming the basic software components in Windows platform. These binaries are in the format of portable

executable (PE). This format is a data structure that contains the information for the loader of the OS to manage the executable code encapsulated inside. It contains code, dynamic library references, API import and export tables, thread local storage (TLS) data and resource data. This format is used for executables, dynamic link libraries (DLLs) and device drivers, etc. The PE format is a modified version of the Unix common object file format (COFF). Usually, a PE file comprises of various headers and sections used for mapping the file into the memory by the dynamic linker. The sections are specified to contain data or code, and can be mapped as either execute/read/write or not. One of the sections is the import address table, which contains APIs used by this binary from other binaries.

Binary Loading. Programs commonly load binaries in addition to its own executable. In Windows, such binaries come in various forms such as DLL, DRV (driver), CPL (control panel), etc. We classify binary loading behaviors into system loading and program loading. System loading describes the loading by the operating system to load the program main executable and the loader component. Windows loads the loader component, NTDLL.DLL, for every its programs, as such it is system loading. Program loading describes the loading of binaries after the system loading. For example, when IE needs an ActiveX control, it loads the binary dynamically after its main executable and loader is loaded. Different operating systems have different APIs used for loading binaries. Windows offers the *CreateProcess* family of APIs to perform system loading. Programs invoke the *LoadLibrary* family of APIs to load binaries, afterwards. In Unix-like systems, programs invoke *dlopen* and *uselib* to perform program loading. The *exec* family of functions is used to load executables together with the loader (*ld.so*). Many of these APIs support loading binaries with either a full path or a relative path. When a relative path is used in these APIs, there are mechanisms to locate the full path of the binary according to the operating system defined order. This is called "binary search order". Table 2.1 shows the effect of the safe binary search order of *LoadLibrary* versus the default binary search order in Windows. The difference between the orders is that the directory set by the *SetDllDirectory* API is searched instead of the current working directory to give a sense of safety to binary loading.

Other than the search order, various settings can also affect the loading behaviors. In Win-

Table 2.1: Binary Search Orders of `LoadLibrary`

`LoadLibrary` Safe Order	`LoadLibrary` Default Order
1.main program directory	1.main program directory
2.directory set by `SetDllDirectory`	2.CWD
3.system and 16-bit system directories	3.system and 16-bit system directories
4.Windows directory	4.Windows directory
5.Environment Variable `PATH`	5.Environment Variable `PATH`

dows, there are many factors affecting the binary loading. System settings, such as environment variables and registry keys, also affect the binary loading behaviors. In program loading behaviors, the factors can be quite arbitrary, for example, the program may use file names and paths from configuration files. We can characterize these factors into four general types:

- system settings, e.g., search order;
- environment settings, e.g., environment variables, registry keys, etc.;
- binaries, e.g., a binary can cause other binaries to load; and
- data used to load binaries which can come from files, e.g., configuration files.

The loading behavior of a binary can be directly or indirectly caused by a factor, e.g. a binary is loaded because its path is in the search order. For example, a configuration file contains the name of a binary for a program to load, and the location of this configuration file is stored in a registry key. The loading behavior then is directly caused by the configuration file, and indirectly caused by the registry key.

2.1.1 ActiveX

ActiveX is a framework for defining reusable software components in a programming language independent way in Windows. It was introduced in 1996 by Microsoft and is commonly used in its Windows operating system. Many Windows applications, such as Internet Explorer (IE), Microsoft Office, Microsoft Visual Studio, and Windows Media Player, use ActiveX controls to build their feature-set. They also use these controls to encapsulate their own functionality that can be embedded into other applications. ActiveX controls are widely used in the web browser application, such as IE. IE allows embedding ActiveX controls onto web pages to enhance the

functionalities from the website. Examples include customized applications for gathering data, viewing certain kinds of files, and displaying animation.

ActiveX Security. Digital signature is the basic security mechanism in ActiveX itself. Digital signing tells users where the control came from and verifies that the control has not been tampered with since its publication. Killbits, which are simple registry flags that instruct the browser not to load the specified control, is another ActiveX security mechanism. The advantage of killbits is that they can easily be set with a simple registry modification. Implementing the IObjectSafety interface with *Safe for Initialization* and *Safe for Scripting* properties is an ActiveX security mechanism checked by the browser. Before loading the control, the browser will verify if the control is safe for initialization and scripting.

IE has an Add-on manager to manage the usage of the ActiveX controls and the safety of these plugins. This manager controls how an ActiveX functions through custom level settings. Many more mechanisms are introduced when updating the browser. ActiveX opt-in deactivates most ActiveX controls by default. When the user encounters a webpage with a disabled ActiveX control, the user can choose to enable the ActiveX control from the prompt information bar. Trusted sites let user to set web site where ActiveX controls can be used freely without any prompt. Per-user ActiveX assures that not only administrator but also common users can install ActiveX for their own profile. Per-Site ActiveX helps preventing malicious repurposing of controls, where an information bar appears asking the user whether or not the control should be permitted to run on the current website.

ActiveX security mechanisms are based on the trust in the controls. These mechanisms do not guarantee the safe usage of the ActiveX control. Even if the designer claims the safety of the controls, it may not be safe. Browser settings involve user interactions to specify whether an ActiveX control is trusted. This pushes the decision making of security to the users, which is not reliable. In addition, these mechanisms either allow the usage of ActiveX controls with full trust or completely deny the usage if the trust is not given. In summary, both the ActiveX control side mechanisms and the browser side mechanisms are based on the user trust and have a coarse grained granularity of control.

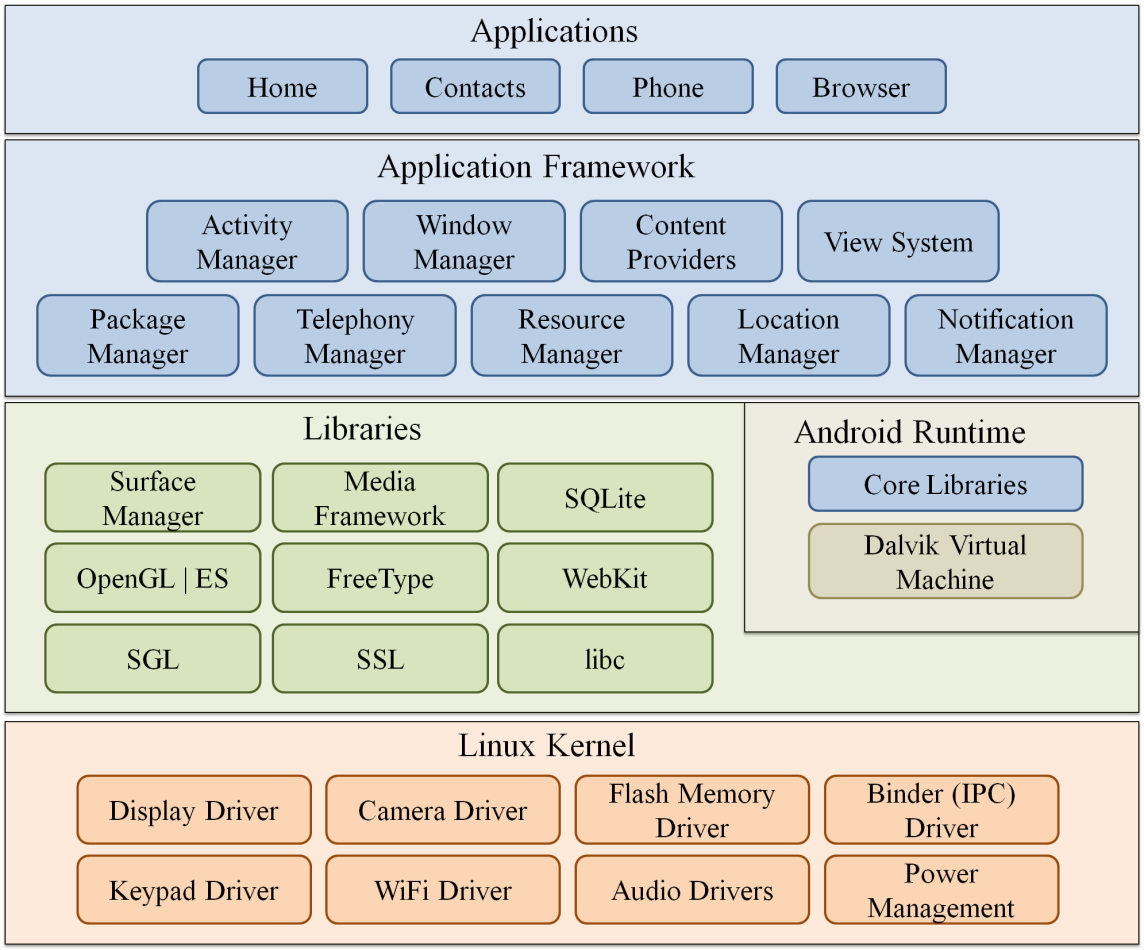


Figure 2.1: Android Architecture

2.2 Android

In this section, we introduce the basics of Android platform and its security mechanisms. Android has a heavy use of component interactions, and it has specific rules on how components interact with each other. We first introduce its architecture, permission model, component types and component interactions. Most of the discussion in this part is adopted from the official Android resources [2]. Then, we discuss the existing works on Android component interactions.

Android Architecture. As illustrated in Figure 2.1, the Android software stack layout has four levels, operating system, middleware, application framework and applications (apps). Android is a Linux kernel-based platform. On top of the kernel is the middleware. It contains various native libraries written both in C/C++ and Java, and Android runtime, which includes the Dalvik Virtual Machine (DVM) similar to Java Virtual Machine (JVM). The libraries at this layer provide various accesses to native functionalities, such as OpenGL and libc, for the

upper layer. At the application framework layer, Android provides well-defined interfaces for Android apps to manager various system resources, such as GPS locations and contact lists. At the application level, each Android app runs in a separate DVM. Android apps are written in Java, and some may contain native code, using Java Native Interfaces (JNI).

Android Permission Model. The Android operating system is based on Linux. Each app is a different user with a unique Linux user ID. The system implements the principle of least privilege by limiting each app within a sandbox[1]. The main security feature of Android platform is provided through the Android permission mechanism. It enforces access controls on the specific operations that a process can perform, and also enforces per-URI permissions for access to specific pieces of data. The basic point of the Android security architecture is that no app has permission to perform operations that can adversely impact other apps. Apps must explicitly share resources and data by declaring the relevant permissions in a manifest file. When installing the app, users decide whether to grant the permission in the manifest file, or deny the request as well as the installation. The enforcement of permissions happens at the process level, where the system will invoke APIs to check the permissions of the operations.

Android Components. Android apps are made of four types of Android components, with each also working as an entry point to the app. Each type of the components is designed with a distinct goal and a life cycle specifying how the component is created and destroyed.

Activities: an *activity* is a single screen with a user interface to offer a specific functionality usually requires user interactions. For example, the screen to take photos is a single activity in a camera app. They can be started individually, and can work together with each other to form a cohesive user experience.

Services: a *service* runs in the background to perform long-running operations or to perform work for remote processes. It does not have a user interface. It is quite similar with background service programs in desktop platforms. For example, checking incoming emails in the background is usually implemented as a background service.

Content providers: a *content provider* manages the set of app data which can be shared with other components or apps. The data is stored in any form of persistent storage locations

[1]This is Android-specific sandbox using Android permissions, unlike Unix ones.

on the device or on the web. Other components can query and modify the data through the content provider component with a proper permission. The contact list in the Android phone is a typical system content provider.

Broadcast receivers: a *broadcast receiver* responds to system-wide broadcast announcements. These broadcasts can be from the system or the apps. It does not create any user interface, but can create a status bar notification instead. Usually, these receiver components work as a "gateway" to other Android components. For example, email managers can register broadcast receivers to be informed when there are incoming email notifications.

Android Inter-component Communication. Android inter-component communication (ICC) is a unique component interaction that allows one app to start another app's component. For example, suppose a user want to use an SMS app to send a photo. Instead of developing an activity to capture a photo in the SMS app, the SMS app can simply start the activity in the camera app for capturing photos. After the photo is captured, it is returned to the SMS app for sending. From the user's point of view, it seems that the camera is actually part of the SMS app. The design goal of the Android ICC is to share system resource and app functionalities.

When the system starts a component, it starts a process for that app (if it's not already running) and instantiates the classes needed for the component. Apps do not directly start another component. The Android system can activate a component in another app instead. One app must specify an "intent" to start another component sending to the system. Intents bind individual components to each other at run time. It defines a message to activate either a specific component (explicitly) or a specific type of component (implicitly).

Unlike the other three types of components, content provider is not activated by intents. Rather, it is activated when targeted by a request from a ContentResolver. The content resolver handles all direct transactions with the content provider so that the component that's performing transactions with the provider invokes methods on the ContentResolver object.

2.2.1 Android Security

Android platform is a component concentrated environment. Android ICC allows permission re-delegation, where an Android app requests another suitable app to deal with certain tasks

using the permissions from the suitable app on its behalf, e.g. through an intent. Because of the misuse in these interactions, privilege escalation can happen to affect Android apps or even the system, when the request handling app gains the privileges (permissions) of the requesting app.

Permission Re-delegation Analysis and Mitigation. One approach to analyze these problems is running static analysis. ComDroid [42] uncovers possible unintended consequences of exposing certain app components. It only detects vulnerabilities within the components. Woodpecker [65] finds all paths and prunes infeasible paths using reachability analysis. It has similar limitations as the ComDroid by finding only vulnerabilities within certain components. CHEX [85] additionally detects cross-component vulnerabilities. SEFA [123] uses provenance analysis to determine the impact of vendor customizations on security of the Android platform and cross-app vulnerabilities. It is similar to Woodpecker and CHEX. These static approaches do not handle dynamic codes in apps, which makes them incomplete in identifying permission re-delegations. Most of them focus on the problems within the components, ignoring the interactions among the components.

To mitigate the potential threat caused by permission re-delegation, run-time approaches [36, 51, 57] either check IPC call chains or monitors the run-time communication among apps. Saint [97] examines the interfaces one app exports to another and extends the Android framework to enforce inter-app security policy at install and run time. Meteor [29] explores the security of multi-market app ecosystem and proposes a new app installation model to retain the original single-market security semantics, e.g. kill switches or developer name consistency. Other approaches use virtualization techniques [24, 80] to solve the problem. These works will be discussed in Chapter 5. MoCFI [50] implements a control-flow integrity enforcement framework for apps but on iOS platform.

Permission Usage Analysis. Many approaches attempt to provide a better understanding of how Android permissions are used. These studies help to identify the Android apps that can potentially cause privilege escalation. VetDroid [131] uses dynamic taint analysis to check how Android apps use permissions to access system resources and how these permission sensitive resources are further used in the apps. Pegasus [41] uses static analysis to build a permission

event graph, which shows the permissions needed by an app from one state to another state. Vidas et al. [116], PScout [25] and Stowaway [56] all study the problem of over-privileged third party apps and provide permission mappings to the system. Barrera et al. [30] study the permission usage patterns of third party apps by applying self-organizing maps. There are works focusing on inferring certain security-related properties about apps based on requested permissions. Kirin [54] looks for hard-coded dangerous combinations of permissions to warn the user about potential malware. Sarma et al. [106], Peng et al. [103] and Chakradeo et al. [39] use machine learning techniques to automatically classify apps as potentially malicious based on the permissions they seek.

Permission System Enhancement. Many approaches focus on extending the Android permission system to enforce fine-grained control at run-time. These approaches are such as allowing a user to authorize partial permission and revoke granted permissions on demand [93]; enforcing usage control based on system context, e.g. device location [28]; offering a privacy mode to enable dynamic management of privacy-related permissions [132]. These approaches [28, 93, 132] generally require the users to have a good understanding of permissions as well as policies. Other approaches focus on enhancing install-time permission assignment such as: detecting undesired combination of requested permissions [32]; mapping Android API to permission labels and detecting permissions over-privilege [33]. However, malicious apps may request permissions that may not trigger such detection systems. Crussell et al. [45] enforce control on both install-time permission assignment and run-time permission usage; but it also imposes on the app developer to specify policies, which requires a large amount of effort and the policies can be error prone.

2.3 Vulnerability Detection Techniques

Both static and dynamic analyses are applied to vulnerability analysis of software programs. Static analyses [22,44,58,81] are usually conservative and sound but less precise than dynamic analyses and testing [90, 94, 111, 117]. Their complementarity is well studied [55, 64], and they can also be combined [64,66]. Static analyses [22,44,58,81] analyze a program statically

without ever executing the program. In contrast, dynamic analyses [38, 62, 63, 73, 90, 94, 111, 117, 128] are typically based on runtime tracing and are precise. Especially, they usually do not require source code of the analyzed program, and can deal with many dynamic program features, e.g., dynamic code. In this section, we focus on what has been found to be generally effective approaches, namely, fuzzing and dynamic program testing. They are related to our techniques in identifying a component misuse vulnerability. Furthermore, the exploration testing discussed in Chapter 3 is inspired by these techniques.

Vulnerability detection techniques have been well explored. In general, most of these approaches focus on detecting buffer overflows, memory corruption errors and integer overflows [22, 34, 37, 44, 58, 81, 90]. These approaches are proved to be effective in detecting such vulnerabilities; yet, they do not focus on the vulnerabilities that are through the normal functionality provided by the software programs, which is a typical cause of the component misuse problem.

Fuzzing and program testing techniques [60, 114, 127] are widely applied to find security vulnerabilities in software programs. These techniques can be applied easily to already deployed software [89]. Fuzzing techniques have successfully revealed many severe software vulnerabilities. The main idea of these techniques is to test the program with specific input data, or even malformed data, and check whether some behaviors are triggered or if the program crashes. Fuzzing and program testing techniques can be applied not only in detecting fault and defect of program itself, but also in security applications. For example, fuzzing can be used in intrusion detection systems [72], web application scanning [31, 68] and even analyzing the configurations of software components [47].

Modern fuzzing and program testing techniques cooperate heavily with other program analysis techniques to improve the result of the analysis. One direction is to apply symbolic-execution to make the fuzzing and testing more intelligent. This idea is widely implemented in a large amount of research work. For instance, DART [62], SAGE [63], and EXE [38] are based on dynamic symbolic execution. They use symbolic values instead of concrete input data to produce abstract conditions over program variables that the concrete tests satisfy during the execution. They have been effectively applied to applications with structured inputs. Godefroid

et al. [61] apply a similar technique with input symbolic grammar specification in detecting vulnerabilities in many shipped Windows applications. Another direction is to apply taint-analysis to direct the fuzzing and testing process, which also makes the convergence process of finding specific input quicker than using simple testing techniques. BuzzFuzz [59] applies taint analysis, which requires program source code. TaintScope [117] is another application of this idea to detect overflow vulnerabilities in programs, which does not need source code.

Fuzzing techniques not only detect vulnerabilities, but also find and help to recover file contents and file formats. File formats fuzzing [113] can be applied to improve identification of configuration files among all the files accessed in a program. Tupni [46] is a tool to reverse-engineer an input format with a set of information based on the observation that applications usually process iterative data records in loops. It can identify record sequences and types in input data, and even detect different types of constraints on the values. Our testing strategy discussed in Chapter 3 for detecting the binary loading vulnerability is inspired by such ideas.

2.4 Execution Paths and Attack Paths Analysis

In this section, we survey different techniques to detect and analyze hidden software vulnerabilities. They are related to our detection mechanism discussed in Chapter 4. These works focus on analyzing program execution paths and attack paths. Analyses in program execution paths focus more on the completeness of behaviors in program execution, whereas analyses in attack paths and graphs focus more on the reachability of different parties involved in certain system interactions.

Analyses in execution paths [92] and code paths [115] provide a more complete picture of the software actions, and even indicate the specific environmental cause of the vulnerabilities. These analyses explore multiple execution paths and identify hidden behaviors when certain conditions are met.

Analysis in attack paths from attack graphs is a generic method to provide security assessment of the computer systems, especially computer networks. One direction is to study the system state, when analyzing attack graphs. Hierarchical aggregation [95] of elements is con-

sidered in these attack graphs. Graph based model checking [108] provides an integral global view of network, where security is assessed by constructing attack graphs in an automated way. Reverse reachability analysis [69] is applied for the safety between servers and clients. Attack tree models [91] are utilized to describe known attacks and test the security properties of protocols, where the attackers are emulated by using a fault injector. Another direction is to study the causality relations between the system conditions in the form of attack dependency graphs [23, 96, 100]. The research work of MulVAL logical attack graph [100] proposes to study the logical dependencies among attack goals and configuration information in identifying security vulnerabilities in enterprise networks.

Our detection mechanism in identifying API-misuse vulnerability is inspired by this area of research, to discover paths on a different level. We focus on detecting attack paths from normal behaviors of the software components.

Chapter 3

Detecting Binary Loading Vulnerabilities

3.1 Introduction

In this chapter, we introduce the problem how various system settings in a complex computer system affect the behaviors of the software components, which can lead to security problems such as binary loading attacks. We propose a solution to detect the dependencies in these component behaviors on the system settings and identify the security impact caused by the misuse of software components. The component misuse discussed in this chapter is insecure binary loading, or binary loading attacks.

Binary loading is a basic operating system functionality that is extensively used in software components. To run a program, the operating system loads the program's binary components into memory, preparing them for execution. A running program, can also dynamically load libraries to extend its functionality. We use binary to refer to the main executable of the program and any other binaries which it loads. In Windows, binaries are dynamic link libraries (DLL), drivers, control panel applets, etc.

Binary loading gives flexibility to software developers and the operating system but it also leads to opportunities for attackers to exploit a program. If attackers can get their malicious binary to be loaded, they can get *arbitrary code execution*. Hence, a binary loading vulnerability is often a critical vulnerability.

Binary loading is affected by many factors in complex computer systems, such as system

settings and program configurations. For example, the search path in Windows directly controls how the absolute path of a dynamic-loadable library (DLL) is resolved. Based on the order of directories in the search path, Windows searches those directories for the desired DLL. Such factors are popular targets used by attackers to get an arbitrary code execution exploit. By the first quarter of 2012, Microsoft published 27 patches [3] for binary loading vulnerabilities. Furthermore, a third-party advisory [4] identified more than 334 vulnerabilities across 122 vendors in Windows applications. Related problems [48,98,99] are found in Linux and Android [104].

To prevent malicious binaries from being loaded, one solution is to identify or confine untrusted binaries [101, 124] but this does not *prevent* unexpected binary loadings. A better way is to identify vulnerabilities in binary loading behaviors. Several solutions [76–78] analyze how a program resolves path names using static or dynamic analysis. They focus on analyzing the result of path resolution process for each binary loading behavior in the system. However, path resolution is only one factor affecting binary loading behaviors; these methods can fail to detect vulnerable loading behaviors relying on other system factors. To better understand software dependencies, data dependency analysis [26] and dynamic tainting analysis [94] have been proposed. However, these dependency analyses do not handle the complexity of software binary loading behaviors. For example, binary loading on Windows is commonly controlled by a combination of configuration files and registry keys.

In this chapter, we introduce a novel approach, LDRSCOPE, which can comprehensively explain binary loading behaviors in Windows in a simple fashion. The core of our approach combines data dependency and system dependency to identify the factors affecting the loading behaviors. Based on dynamic program behaviors recorded during execution, the combined dependency analysis accurately identifies all the attacker-controllable factors of a binary loading. LDRSCOPE generates a loading dependency graph (LDG), which clearly identifies the factors that cause and control binary loading and also flags vulnerabilities detected. What distinguishes our work is the emphasis on understandable explanations – explaining the interactions of binary components and system settings that control binary loading. One reason why binary loading vulnerabilities are common is that software developers do not fully understand the system in-

teractions, which can involve more than just their own code. It requires a comprehensive understanding of Windows, such as how various types of Windows libraries and third party libraries work and interact with one another. In addition, configuration files may also result in binary loading behaviors, which are complex and can be misunderstood by the developers. Our graph explanations and vulnerability analysis fill this gap.

We evaluated a prototype of LDRSCOPE in Windows on 34 real-world applications finding 24 binary loading vulnerabilities with attempts to load missing binaries. We identify 9 applications using configuration file unsafely, and 3 of them lead to loading vulnerabilities caused by configuration files. Among the loading vulnerabilities, LDRSCOPE precisely identifies the binaries that need to be fixed. We present several use cases of the LDG to help software developers. Using our approach, we not only provide precise analysis on how software component behaviors are affected by system settings and other aspects of a complex computer system, but also help the developers fixing their applications in loading binary components.

3.2 Binary Loading

Loading behaviors are program behaviors that invoke APIs (explicitly or implicitly) to load binaries. Windows offers the `CreateProcess` family of APIs to load the executables and `ntdll.dll`. Programs invoke the family of `LoadLibrary` functions to load DLLs. Many of these APIs support loading binaries with either a full or relative path. When a relative path is used, there are mechanisms to locate the binary using a set of search paths according to OS-defined order, which is called *binary search order*. For example, in Windows, `LoadLibrary` with the default search order searches the following locations: main program directory, current working directory (CWD), system directories, Windows directory, and directories in the environment variable `PATH`. Developers can apply a "safe binary search order" using `SetDllDirectory` to change the second location CWD to their directories.

We call the underlying reasons that determine how and why binaries are loaded as *loading factors*. Typically, a loading factor comes from the environment that the program runs in. For example, a string in the executable specifying a DLL can lead to the DLL being loaded; hence,

the executable becomes a loading factor. However, the loading might also be caused by data that comes from the command line or a registry key. Changing the loading factor will affect the binary loading behavior. We summarize the typical loading factors in Windows (abstractions of actual operating system objects or entities): file factors are data files; registry factors are registry keys; binary factors are binaries; and program environment factors (PEB: Program Environment Block) which is a special data structure in a Windows process. The PEB helps interface the process with the Windows operating system and includes the current working directory, main program directory, system directories, environment variables, command-line inputs and binary search order. A detailed discussion on binary loading and the loading factors can be found in Section 2.1.

3.2.1 A Motivating Example

The loading behavior of programs can have complex interactions between the loading factors. We illustrate the problem of binary loading with an example.

IDA Pro is a commercial disassembler and debugger. It loads a graph rendering component specified by the `GRAPH_VISUALIZER` option in a default configuration file `ida.cfg` to visualize flow charts. This setting can be overridden by an optional user configuration file `idauser.cfg`. The `ida.cfg` configuration file is safe as it is usually not controllable by the attackers. However, IDA Pro searches and loads the optional user configuration file from the current working directory. This may result in the following binary loading attack (in version 5.5):

- An attacker sets `GRAPH_VISUALIZER` to a binary payload `d:/malicious.exe`;
- The attacker places both the configuration file `idauser.cfg` and the payload in a network shared folder, e.g. `d:/` in the victim's system;
- IDA Pro is started in `d:/` by opening a supported file which uses the graph rendering component;
- The binary `malicious.exe` is loaded and executed.

Here, among others, the critical loading factor is the configuration file `idauser.cfg`, a clear identification and understandable explanation of loading factors will show that the software has an insecure feature, which will either need to be modified or disabled. It also illustrates that software developers need to understand software configuration issues. Tools to identify and explain the issues will lead again to more secure program development.

The example illustrates the complex behaviors, which can lead to complex interactions with loading factors, and give an idea why such vulnerabilities may be lurking in the codebase. It is not easy for software developers to even realize that they have binary loading vulnerabilities, let alone to fix the problem.

3.2.2 Loading Dependencies

From analyzing the above example, we learn that loading factors, e.g. configuration file `idauser.cfg`, determine the loading of the malicious binary `malicious.exe`. IDA Pro reads the data (the string `GRAPH_VISUALIZER=d:/malicious.exe`) from the configuration file to the memory buffer. It parses this memory buffer and calls `LoadLibrary` to load the binary. The data is copied from the file (loading factor) to the memory and finally passed to the loading API as an argument, which we call the *loading argument*. For example, a typical loading argument in this example is `d:/malicious.exe` (a string in the memory) used by the `LoadLibrary` API. We can see the loading factors affect the loading behaviors through propagating data in memory. The copying or passing of data in memory involves instructions, such as `mov`. Figure 3.4 gives more details of this example. Here we briefly describe features of the example:

- The file data is first read to the memory region at `0x008fd620`;
- Then, the file data is copied to the memory at the address `0x0089ba8a`;
- Finally it is passed in the register `eax` as the argument of a loading API invocation (*call [API_OFFSET]*).

Thus, there is a data dependency from the loading argument to the loading factor at the instruction level. To understand how and why a loading behavior happens, we need to recover

this data dependency by tracking back from the result (loading behaviors) to the cause (loading factors). Intuitively, a straightforward solution is to apply some form of data dependency analysis [26] on the loading argument at the instruction level.

A memory region, such as `0x008fd620`, recovered from the data dependency analysis is not a useful explanation of loading behaviors. There exist other dependencies from the loading behaviors, which cannot be resolved alone by the data dependency analysis. In the above example, IDA Pro reads a default configuration file for the user configuration file path `d:/idauser.cfg`, and from this string, it reads the file into the memory region at `0x008fd620`. There is a dependency from the memory region representing the file buffer of `idauser.cfg` to the string representing the file path `d:/idauser.cfg` of `idauser.cfg`. This dependency is not directly through data. Instead, it is at the system level, which we call "*system dependency*". To detect these dependencies, we need a different form of analysis other than the data dependency analysis. Thus, we introduce system dependency analysis, which investigates various entities at the operating system level, such as file handles (i.e. Windows equivalent of Unix file descriptors). Direct data dependency analysis can link the loading argument `d:/malicious.exe` to a memory region at `0x008fd620`. System dependency analysis then resolves this memory region to a file buffer and links it to a string `idauser.cfg` representing the file path in the memory. Subsequent data and system dependency analysis can recover that this file path is dependent on another default configuration file `ida.cfg`. Step by step, all the dependencies of the loading of the binary `malicious.exe` can be fully recovered by these analyses to find all the loading factors in an execution.

In this chapter, both data dependency and system dependency of the loading behaviors are called "loading dependencies". We combine data dependency analysis and system dependency analysis as "loading dependency analysis". It provides a more comprehensive explanation of the loading behavior to the developers, where all the loading factors either directly or indirectly affecting the loading behaviors are shown. The tool we developed is LDRSCOPE. It has the following design objectives:

- **Comprehensive Explanation of Loading Behaviors.** The comprehensiveness in this chapter does not mean the approach is complete, rather, it describes that our approach

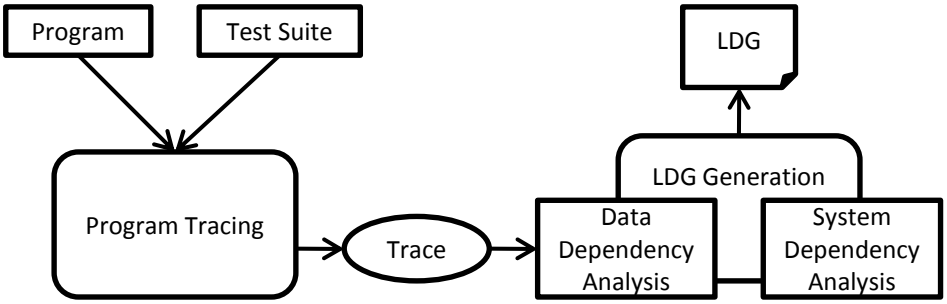


Figure 3.1: Structure of LDRSCOPE

not only detects loading behaviors, but also explains why and how the loading behaviors happen. The reasons or the factors should be easy to understand as we described in the previous discussion.

- **No Source Code Requirement.** Due to the closed source nature of Windows system, we assume the binaries to be analyzed do not come with any additional information, e.g. symbol tables. This means no complete source code is provided. Although developers can use their own source code to analyze their binaries, this is not required for a typical binary analysis. In summary, given a binary without additional information, LDRSCOPE can perform dependency analysis correctly. Dynamic analysis is more effective to be applied to this problem than static analysis. Without source code, static analysis faces a challenge of correctly disassembling the binaries, which also contains many dynamic program features, e.g. dynamic code. In Windows, system features such as heavy use of kernel callbacks further complicate static analysis. This usually leads to either accuracy problem or effectiveness problem.
- **General Usage and Visualization.** It should provide general usages for the developers to understand the loading behaviors. It should also be able to identify the binary loading vulnerabilities we discussed in the previous sections. It should provide a readable explanation in the form of visualizations showing how the loading behaviors come up in the system. This is useful for both the developers and security analysts.

3.3 Design

We now describe the design of LDRSCOPE, whose goal is to explain binary loading behaviors and evaluate the security of binary loading in Windows. In the previous discussion, we attribute the binary loading behaviors to loading dependencies. The key to solve the binary loading problem is to identify these loading dependencies. We define a loading dependency graph to visualize these loading dependencies. To identify these dependencies, we apply dynamic dependency analysis of both program data, i.e. memory, and system states in LDRSCOPE. Overall, it contains two stages, program tracing, and loading dependency graph (LDG) generation shown in Figure 3.1. LDRSCOPE first applies dynamic program tracing on the program where test suites exercise its normal behaviors. It records the program's execution and generates an instruction trace. From the trace, various system information, such as binary loading behaviors and operations on the system objects, e.g. opening and reading of files, is extracted by identifying system calls and system APIs. Then, LDRSCOPE applies and combines data dependency analysis and system dependency analysis on the data collected in the program trace, and outputs LDGs to visualize the loading dependencies–the dependency from the loading behaviors to the loading factors. Although the approach in this chapter focuses on Windows, the core design ideas can be modified for other operating systems. We describe what is an LDG and how the LDG is generated, and finally show how loading behaviors can be studied through these LDGs.

3.3.1 The Loading Dependency Graph

To provide comprehensive understanding of program loading behaviors, we introduce loading dependency graphs. It visualizes why and how a loading behavior happens. An LDG is a directed acyclic graph showing the loading dependencies. We define a loading dependency graph G as an ordered pair $G = (V, E)$ where

- V is a set of nodes. More specifically, V contains V_a and V_f, where V_a are nodes representing the arguments to loading APIs and file pathnames, and V_f is are nodes representing the loading factors.
- E is a set of edges. More specifically, E contains E_d and E_s, where $E_d \subseteq V_a \times V_f$, rep-

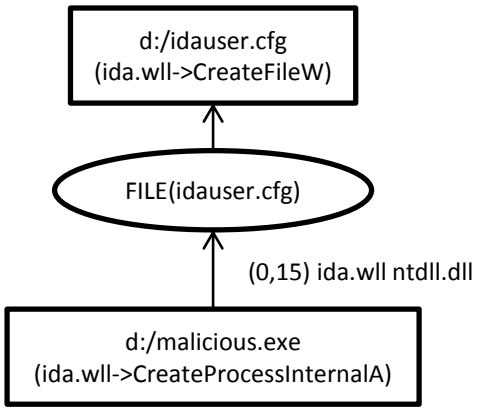


Figure 3.2: A Simple LDG

resenting the data dependency, and $E_s \subseteq V_f \times V_a$, representing the system dependency.

Figure 3.2 shows a simple example of an LDG. The elements of an LDG are as follows:

- Rectangular nodes denote arguments to loading APIs, such as binary pathnames (e.g. `d:/malicious.exe`) and file pathnames (e.g. `d:/idauser.cfg`). The target binary is the root node (the node at the bottom). The label of rectangular nodes shows the binary and the API used to access the argument, e.g. label `d:/idauser.cfg (ida.wll→CreateFileW)` indicates `ida.wll` invokes the `CreateFileW` API to access the file `d:/idauser.cfg`. A notation `loader→sysload` is used to represent the system loading, where the OS kernel loads the program main executable and the loader component. For example, the `ntdll.dll` is loaded this way.
- Oval nodes denote loading factors. The label of oval nodes shows the type and value of the loading factor, e.g. `FILE(idauser.cfg)` indicates a loading factor, which is FILE type loading factor, and the value of it is `idauser.cfg`.
- Edges with arrows denote the loading dependencies between nodes. The label of an edge shows which characters of the string in the rectangular node argument are used in the data dependency and the binaries of which the executed code leads to the data dependency. More specifically, a label $(l, h)B$ indicates values from index l to h of the data is used through the code in binary B. For example, `(0, 15) ida.wll ntdll.dll` shows the string `d:/malicious.exe` is used through the binaries `ida.wll` and `ntdll.dll`.

An IDA Pro Example. A more complicated LDG of the IDA Pro example is shown in Figure 3.3. To make it easier to understand loading dependencies, we use additional markings on the LDGs in this chapter. The dash-lined edges form a path from the loading behavior to a loading factor, indicating the loading behavior is unsafe if this loading factor is controllable by attackers. The grey color highlighted loading factors that are unsafe and conditionally safe (discussed later in Section 3.3.3). The binary that contains the code to load the target binary is also highlighted in grey. There is a box to group binary factors from the same vendor.[1] This LDG shows the loading dependencies of the binary `d:/malicious.exe`. It also gives some short form strings used as macros to shorten values in the LDGs in this chapter. By our design and due to how Windows works, leaf nodes of an LDG can only be BIN, UNK and PEB loading factors. The loading factors are discussed in Section 3.3.2.

3.3.2 Loading Dependency Graph Generation

To extract loading dependencies, LDRSCOPE applies data dependency and system dependency analyses exhaustively on the traces from the program tracing stage, until it identifies all the dependencies for a loading behavior. We call this step, *loading dependency analysis*.

Data Dependency Analysis

To know how loading factors affect loading behaviors, we apply data dependency analysis to identify the loading factors directly affecting the loading behaviors in the traces collected from program tracing. This involves identification of loading arguments and backtracking the loading arguments from the effect to the cause (loading factors). During the identification of loading arguments, we first find the invocation of the loading APIs by checking the address of that API. We extract the loading argument from the point where invocation of the loading API happens. The loading argument is in the form of a memory region M_t in the trace. Then, we backtrack through this memory region M_t based on the data dependency. In the trace, an instruction I can have a source operand s and a target operand t whose effect is that values from s are written to t. The backtracking first locates the last instruction I_n whose target t_n is M_t.

[1] Abridged LDGs omit this grouping of binaries and the binary involved in the data dependency on the label of edges. We use abridged LDGs in the evaluation section to omit the details, where they are not discussed.

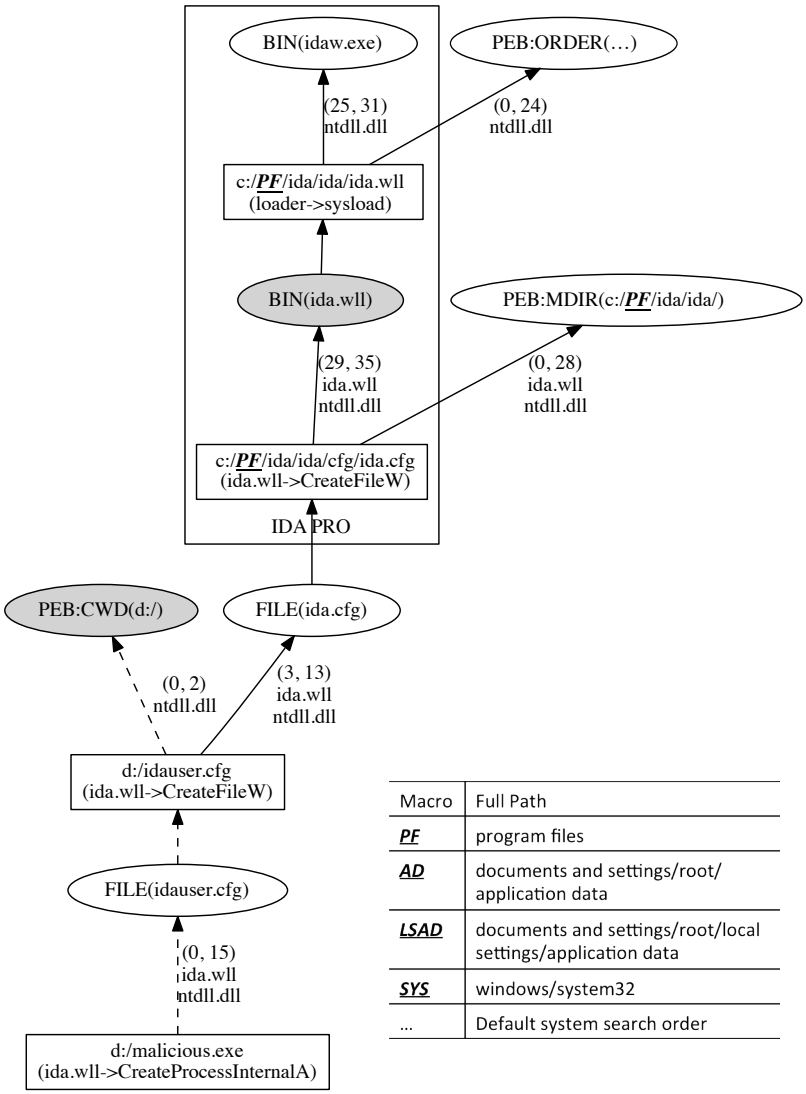


Figure 3.3: Loading Dependency of `d:/malicious.exe`

Then, it searches backwards in the trace for the closest instruction I_{n-1} such that t_{n-1} equals s_n. The backwards search is repeated recursively until there are no more instructions in the trace whose data dependency can be followed. And the source memory region M_s is recovered from the source s_1 of the first instruction in this backtracking.

As shown in Figure 3.4, we illustrate the basic operation of dependency analysis with a simplification of the IDA Pro example in Section 3.2.1. Suppose in the execution, the `LoadLibrary` API is called with the loading argument `d:/malicious.exe`. It resides in a memory region starting from the address `0x001489a2`. Each node in the figure uses the following notation to describe the execution of the instruction extracted in the trace–[*ic*] *pc*: *instruction*:

- *ic* is the instruction counter

Table 3.1: Loading Factors for Windows

Type	Description	Value	Memory Region
FILE	files	file path name	file buffer
REG	registry keys	key name	key value buffer
BIN	binaries	binary path name	binary section
PEB:ORDER	binary search order	path names	name buffer
PEB:MDIR	main program directory	path name	name buffer
PEB:CWD	current working directory	path name	name buffer
PEB:SYSDIR	system directories	path name	name buffer
PEB:ENV	environment variables	variable pairs	variable buffer
PEB:CMD	command-line inputs	input values	string buffer
UNK	untracked memory region	memory address	memory region

- *pc* is the program counter
- *instruction* is followed by the details of its arguments. In the node, the bottom lines show the detail of each operand. The form is *type* "at" *addr*[*value*](*access*). *type* indicates whether the operand is a register or a memory value. *addr* shows the memory address of the operand, if it is a memory value, or the name of the register, if it is a register. *value* shows the data in the operand. *access* indicates how the operand is accessed, either read/write or both.

The value `0064` stands for the Unicode 'd', which is the first letter in the loading argument. The bottom node (the 259311580*th* instruction in the trace) is a `mov` instruction. It reads the value `0x64` ('`d`') from the register `ax` and writes to address `0x001489a2` (whose original content is `0x4e`). Therefore, there is a data dependency to the instruction in the middle node (the 259311577*th* instruction), which depends on the upper node (the 259309246*th* instruction). Thus, the first letter of the loading argument `d:/malicious.exe` is traced back to the `rep movsd` instruction and the memory region starting from `0x008fd620`. The dependency between the final target (bottom) and original source (top) is shown in Figure 3.5a.

System Dependency Analysis

As discussed in Section 3.2.2, we also need to deal with the indirect dependency from the data to the binary loading API in a program. The pure data dependency analysis leads us from a memory region to another region, which does not provide a comprehensive way of loading

Algorithm 1 Hybrid Dependency Analysis

procedure ADDSOURCEOPERAND($insn$)
  for $operand$ in $insn.src$ **do**
    $ListAdd(operand, list_{var})$ ▷ $list_{var}$ stores the operands with data dependency
    $ListAdd(operand.memreg_{index}, list_{var})$ ▷ for table look-up operations
  end for
end procedure
procedure DATADEPEND($insn_{target}$) ▷ $insn_{target}$ contains the target variable
  $AddSourceOperand(insn_{target})$
  while $GetPrevInsn(trace, insn) = TRUE$ **do**
    for $operand$ in $insn.dst$ **do**
      if $ListLookup(operand, list_{var}) = TRUE$ **then**
        $ListDelete(operand, list_{var})$
        $AddSourceOperand(insn)$
      end if
    end for
  end while
end procedure
procedure ADDRELATEDADDR($list_{var}$)
  for $operand$ in $list_{var}$ **do**
    if $IsRootOperand(operand)\&\&IsMemAddr(operand)$ **then**
      $ListAdd(operand, list_{addr})$ ▷ $list_{addr}$ stores the memory address with data dependency
    end if
  end for
end procedure
procedure SYSDEPEND($addr, table$)
  $obj \leftarrow ResolveVirtualObject(addr, table)$
  if $obj.name\_arg = NULL$ **then**
    $ListAdd(obj, list_{factor})$ ▷ $list_{factor}$ contains all the loading factors
  else
    $ListAdd(obj, list_{factor})$
    $HybridDepend(obj.name\_arg, trace)$
  end if
end procedure
procedure HYBRIDDEPEND($arg, trace$)
  $insn_{target} \leftarrow GetArgInsn(arg, trace)$ ▷ gets the instruction containing memory variable of the loading argument
  $DataDepend(insn_{target})$
  $AddRelatedAddr(list_{var})$
  for $operand$ in $list_{addr}$ **do**
    $SysDepend(operand, table)$
  end for
end procedure

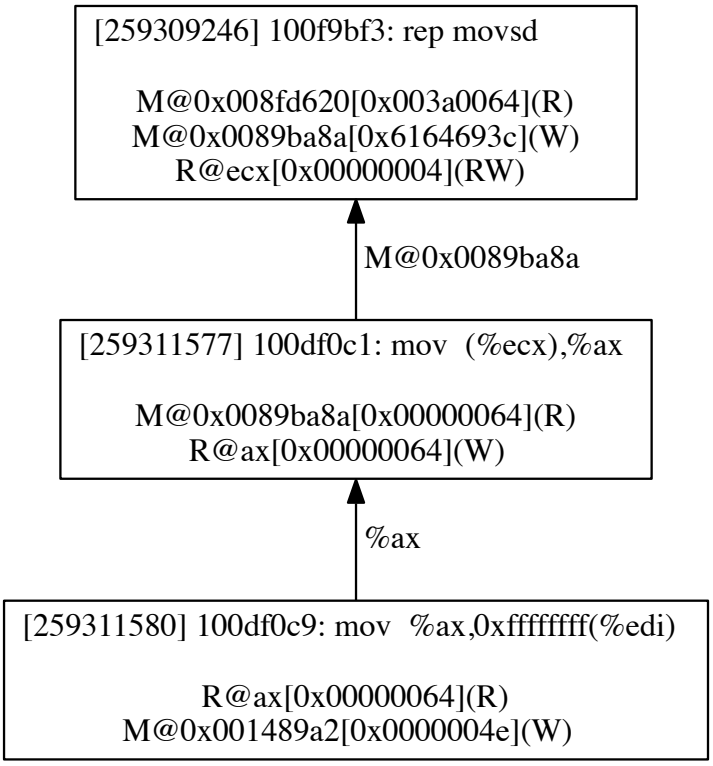
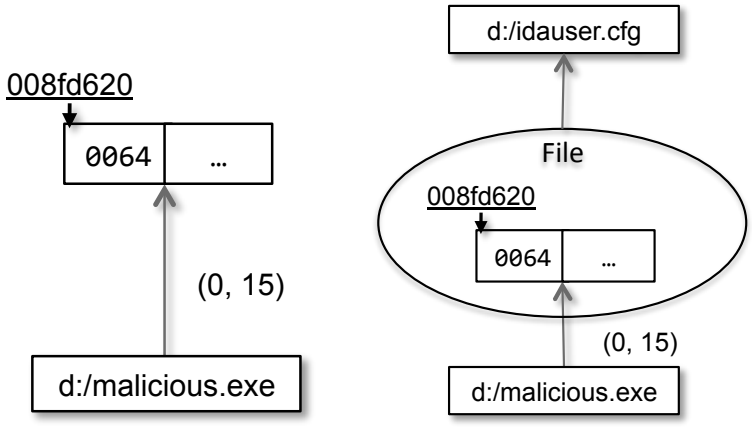


Figure 3.4: Data Dependency Analysis

behaviors. We need the system dependency analysis to explain these memory regions as loading factors and find more links to system settings and configurations.

To understand system dependency, we need a clear definition of loading factors, listed in Table 3.1 with descriptions. Each loading factor has two properties, a *value* and a *memory region*. The *value* identifies the system object that the loading factor represents, e.g. an absolute path name for a file factor. The *memory region* describes the memory region associated to the system object in the process, e.g. a file factor (FILE) has regions of memory as file buffers. In Table 3.1, the untracked factors (UNK) are special factors recovered from the program trace, which represent memory regions for which the origin is outside the program process being traced. For example, process p can write to the memory of process q, which is one way of doing DLL injection using the debugger APIs. From the viewpoint of process q, it will appear as if certain source memory regions cannot be explained, actually it is due to process p but that cannot be explained inside q alone.

System dependency analysis involves resolving of memory regions and linking of the loading factors. During the resolving of memory regions, we resolve the memory region to its



(a) Result of Data Dependency Analysis (b) System Dependency Analysis

Figure 3.5: System Dependency Analysis

loading factor. We extract the system object information from the program trace by investigating the creation of these objects. Suppose in the investigation, we detect that a file with pathname `d:/idauser.cfg` is created and loaded into memory ranging from m to $m + n$. Then, a memory region ranging from $m + i$ to $m + j$ ($0 < i < j < n$) is resolved as a file buffer F of the loading factor FILE `d:/idauser.cfg`. Note that the pathname (usually used by the file creation API) stays in another memory region F_{name}. After that, linking of the loading factors is applied, which is done by linking the *memory region* of a loading factor to its *value*. Continuing the example, the file buffer F (memory region property) is linked to the other memory region F_{name} representing the file pathname (value property). F_{name} and F do not have data dependencies. Their relationship can only be identified by the system dependency analysis. Now, subsequent data dependency analysis can be performed on this F_{name} to recover more loading dependencies.

We describe the system dependency analysis with the same example used in data dependency analysis. The loading argument `d:/malicious.exe` depends on the memory region at `0x008fd620` shown in Figure 3.5a. Firstly, LDRSCOPE resolves this memory region to a file loading factor (FILE) `idauser.cfg` represented as an oval node in Figure 3.5b. The edge label `(0,15)` indicates that the characters from offset 0 to 15 of the string `d:/malicious.exe` (the whole string) depend on this factor. Secondly, LDRSCOPE links this memory region to a file pathname `d:/idauser.cfg` – the *value* property. The result of the system dependency analysis shows that the loading argument `d:/malicious.exe`

indirectly depends on a memory region representing the file pathname `d:/idauser.cfg`.

The loading dependency analysis is illustrated with Algorithm 1[2]. Given a program trace *trace*, a system object table *table* and an argument *arg*, *HybridDepend* extracts all the loading factors that have loading dependency on this *arg* to a list $list_{factor}$. The *table* input is a mapping of memory region to system object, e.g. a file object mapped to a memory region as a file buffer. This table is generated by monitoring the system calls in program tracing. For example, we log the *open* and *read* system calls to extract a file object and map it to the memory regions that are used as file buffers by the *read* system call. We apply this for all the system objects[3] defined as loading factors and construct the system object table *table*. The argument *arg* can be a loading argument, or an argument as a system object for system dependency analysis, e.g., a file object or a registry key object.

More specifically, *HybridDepend* first identifies the target instruction $insn_{target}$ using the helper function *GetArgInsn* in Line 35. *GetArgInsn* extracts the instruction[4] operating on the loading argument *arg* (e.g., `d:/malicious.exe`), right before *arg* is used by the *LoadLibrary* API call. Then, *HybridDepend* invokes the *DataDepend* procedure in Line 36, which backtracks from the target instructions $insn_{target}$ to the source instructions. From Line 7 to Line 17, *DataDepend* first adds the target instructions into the list $list_{var}$ using the helper function *AddSourceOperand*, which puts all the related source operands in an instruction into $list_{var}$. Then, it compares the source and target operands of two consecutive instructions through the trace using the while loop. The checking is made such that the $list_{var}$ always updates itself to the latest source operands with data dependencies. Note that the trace is only traversed once from the target instruction to the beginning to extract the data dependencies in this procedure. After that, *HybridDepend* invokes the helper function *AddRelatedAddr* in Line 37, which checks whether the source operands are already loading factors or need further resolution. If further resolution is needed, these operands are put as memory addresses into a list $list_{addr}$, from Line 18 to Line 24. For each of these memory addresses in $list_{addr}$ shown in Line 38, *Hy-*

[2]This is the pseudo algorithm. The real algorithm includes details on the operations of each byte of the memory variables. We omit those details.

[3]We monitor different system calls for different system objects.

[4]For example, the `push eax` instruction, which pushes the loading argument onto the stack before calling the library loading API.

bridDepend finally invokes the *SysDepend* procedure in Line 39 to apply system dependency analysis. From Line 25 to Line 33, *SysDepend* invokes *ResolveVirtualObject* to resolves the memory address *addr* to a loading factor by using the mappings in the table *table*. Line 27 shows the condition, whether the loading factor is final or leads to any other dependencies. In both cases, *SysDepend* puts the loading factor into $list_{factor}$ as a final result. If the loading factor leads to other dependencies, *SysDepend* applies a new round of loading dependency analysis by invoking *HybridDepend*. Note that the *HybridDepend* procedure recursively invokes itself by calling the *SysDepend* procedure.

The algorithm terminates when all loading factors are retrieved in the list $list_{factor}$. Let *n* be the size of the trace, and *s* be the number of the system objects. The space complexity is *O(s+n)*, since a list of *s* memory addresses is maintained in memory. The time complexity is *O(sn)*, since *s* times[5] of traversal from the target instruction to the beginning of the trace are needed. Note that *s* is usually several orders of magnitude smaller than *n*. Given a trace, the loading dependency generation is optimal since the lower bound of any analysis is to examine the trace.

3.3.3 Loading Dependency Graph Inspection

An LDG helps to understand binary loading behaviors and identify binary loading vulnerabilities. In the threat model of LDRSCOPE, we assume the attackers do not have direct access to the victim's system. Rather attacks[6] exploit loading factors to get malicious code loading/execution. The attackers may also convince users to install (implicitly trusted) programs, which can affect loading behaviors. Based on this model, we classify the loading factors and loading behaviors by the level of threats: *safe*, *unsafe*, and *conditionally safe*. For example, a safe loading factor can be a directory that is private and not accessible to attackers under our threat model, e.g. system directories and default program installation directories. Factors such as `PEB:CWD(c:/system32)` and `PEB:MainDir(c:/program files)` are of this type. Unsafe loading factors include directories that are public and accessible to attackers,

[5]The actual implementation of the algorithm traverses the trace only once, but checks the system object table *s* times. The time complexity is still the same, but the actual implementation is more efficient.

[6]Other attack models are also possible, e.g. local attackers with restricted privileges on the system try to gain the root access.

e.g. shared folders. Factors such as `PEB:CWD(z:/shared)` are considered unsafe. The attacker can plant arbitrary binaries for a vulnerable program to load from unsafe locations. All other loading factors are conditionally safe, which means if these factors are modified to be accessible by the attackers, the factor becomes unsafe. Programmers should be careful about how they use such locations either in code, data or configuration files. LDRSCOPE highlights the unsafe loading behaviors and warns for the conditionally safe loading behaviors. The intent is to alert developers to binary loading vulnerabilities including the possibility that some loading factors such as directories may be accessible by attackers under certain conditions.

We show a detailed analysis on the motivating example in Figure 3.3. We describe the LDG from the bottom node to the top nodes step by step in the following discussion.

- The loading behavior of `d:/malicious.exe` (bottom node) depends on a file factor with the path name `d:/idauser.cfg`. This is the user configuration file.
- The configuration file further depends on CWD factor `PEB:CWD(d:/)` and another file `ida.cfg`. This indicates the use of the user configuration file is in current working directory and is specified by the default configuration file `ida.cfg`.
- The file `ida.cfg` depends on the program main directory `PEB:MDIR(c:/program files/ida/ida/)` and a binary `ida.wll`. This indicates that the default configuration file is used in the program's main directory and is specified by the `ida.wll` binary component.
- `ida.wll` depends on the main program executable `idaw.exe` and the Windows binary search order PEB:ORDER. This indicates that the main executable of IDA Pro loads its component `ida.wll` by finding it according to the binary search order.

This loading behavior is vulnerable, because the loading factor `PEB:CWD(d:/)` (highlighted) is assumed to be unsafe.[7] The dash-lined path indicates the loading behavior can be affected by attackers using configuration file `d:/idauser.cfg` in the CWD `PEB:CWD(d:/)`. The vulnerability is caused by an unsafe use of the configuration file in CWD, just as we analyzed in Section 3.2.

[7] `d:/` is unsafe because in the example setting, this directory is a shared directory that is accessible to attackers.

The above example demonstrates a typical usage of LDG to identify vulnerable binary loading behaviors. Other than that, LDRSCOPE extracts more information into LDGs to provide various use cases for the developers and analysts. We summarize them as follows:

- **Vulnerability Provenance.** A loading behavior usually depends on multiple binaries that can belong to different vendors, e.g. some are shipped with the OS, some are installed with the program and some are added as plug-ins and extensions. One usage of the LDG is to identify which of these vendors cause the loading vulnerabilities. LDRSCOPE groups the binary factors in the LDG by vendors, and attributes the vulnerability to the binaries of these groups. For example, shown in Figure 3.3, the binary `ida.wll` invokes `CreateProcessInternalA` to load the payload, which means the loading code resides in the binary `ida.wll`. Another node shows `ida.wll` also invokes `CreateFileW` to access the configuration file. LDRSCOPE highlights both the binaries that load the payload and access the configuration file. It happens that the two binaries are the same. Then, LDRSCOPE identifies that the problem is within IDA PRO, as `ida.wll` belongs to this vendor.
- **Configuration File Inspection.** As shown in the IDA Pro example, many loading behaviors are exploited from unsafe usage of configuration files. LDRSCOPE can detect the attempts to use configuration files in software components and find the dependencies from the loading behaviors to the data in these files.
- **Potential Loading Detection.** Windows uses the binary search order to locate binaries. When binaries are not present in the places specified in this order, the binary loading can be failed. This failed binary loading is called a *"loading attempt"* in this chapter. This loading attempt can be leveraged by attackers, where they can plant binaries in the places specified in this attempt. Next time, the binary loading will succeed by loading the binary provided by the attackers. LDRSCOPE can detect these loading attempts to warn the users of the potential loading attacks.
- **Loading Behavior Comparison.** The LDGs generated by LDRSCOPE can be used to compare the loading behaviors. For example, they can be used to study the loading

behaviors among different versions of the same software or different software sharing the same components in a software suite.

- **In-depth Explanation.** LDRSCOPE can explain the reasons of the loading behaviors with various factors at system level. Not only the loading caused by configuration files, but also some special loading behaviors caused by injection techniques, e.g. DLL injections. These loading behaviors appear to be misunderstood by the previous solutions [76, 77].

Many of these use cases help the developers to find and fix the problem in their binary components. Section 3.5 gives more detailed scenarios of our analysis.

3.4 Exploration Testing

In this section, we discuss exploration tests and several testing strategies, which can help to effectively apply LDRSCOPE on real world applications and find more binary loading attacks.

3.4.1 Light-weight Testing

LDRSCOPE is based on dynamic tracing at the instruction level through instrumentation on programs using test suites from software developers. As the instruction level tracing is heavy, we can effectively reduce the times of these tracing, if we know whether there are loading behaviors in a program and which test suite can lead to the loading. If this information is acquired, we can directly use the test suite that can trigger the loading behaviors of our interests, instead of running all the test suites in a heavy instrumentation framework. The objective of this testing is to filter out test cases to reduce the tracing workload, if tracing is too time consuming.

For this reason, we apply a light-weight testing based on a monitoring mechanism of system calls. The objectives of the light-weight testing are to detect:

1. whether there are loading attempts inside the program, and if there are;
2. whether the attempt is searching for binaries or configuration files related to binary loading.

3. what is the result of the attempt.

The rationale of these tests is based on the observation that programs attempt to check the existence of certain loading factors, e.g. binaries or the configuration files, before loading the binaries. Attackers can hijack these attempts of loading the binary or the configuration file, and substitute the intended binary by their own ones, especially, when the binary or the configuration files are not existed in a location. There are many places in the system for the attacker to hijack the loading. The current working directory (CWD) is one of these places that are easier to exploit than other locations.

We monitor certain system calls related to the loading factors, e.g. file open, memory mapping and registry operations. From the return values, we check whether certain locations that are susceptible to attacks, e.g., CWD, exist. We also place a payload binary in the location specified in these system calls, and check whether the program loads the payload. The successful loading usually leads to a certain binary loading vulnerability. We can then further run this test suite with LDRSCOPE.

3.4.2 Configuration File Testing

There are loading behaviors through the using of configuration files. Direct testing by monitoring the binary loading system calls is not enough to identify these cases. We apply a heuristic-based strategy to identify whether a configuration file affects the loading and how it affects the loading behaviors. Our assumption of a configuration file is that it is a plain-text file with read and write file permissions; the content of the file contains the configuration settings in the form of "*option=value*".

First, we search for configuration file candidates. We filter out files with known file types that are unlikely to be configuration files when monitoring the file open system calls, such as video or image files. We select the files that can be recognized by the operating system as configuration file, with a specific file type, such as *INI* files in Windows.

Second, we test whether a configuration file setting in the form of *option=value* in these files affect the loading. To achieve this goal, we construct and test these settings in the configuration files. To construct a setting, we need to know the *option* and *value*. We apply heuristics to

select the *option* base on the observation that a configuration option is usually a string stored in the data section of the program binary and used between the "open" and "close" event of the configuration file. We extract the strings meet these requirements as the *option* to construct a setting.

Among these *option* strings, we need to identify whether and which of them is a configuration file setting that triggers a loading behavior by specifying a location as the *value*. We construct multiple settings using these *option* strings and the *value* set to be a fixed location, where a payload binary is prepared. Then, we apply a testing strategy, shown in Algorithm 2 to locate a certain *option* string that triggers the loading in the location. The algorithm begins with the configuration file *conf* containing all the constructed settings. It runs the program and checks whether the options in *conf* can trigger the loading of the payload in Line 2. It splits *conf* into *conf1* and *conf2* if the payload is loaded in Line 6. Each of *conf1* and *conf2* contains half of the options from *conf*. The algorithm continues performing split tests separately for *conf1* and *conf2* in Line 7 and 8. In Line 4, the split test ends, finding all the options lead to the loading of the payload in *r*esults.

The algorithm performs $\log_2 \binom{m}{n}$ times of tests to locate *m* out of *n* options in the configuration file that affects the loading. If there is only one option which triggers the binary loading, only $\log_2 n$ tests are needed instead of testing each *option* string one by one. Although this is a simple strategy, it is sufficient to identify the configuration file options in the real world unsafe binary loadings discussed in Section 3.5.1. There are also many orthogonal fuzzing approaches discussed in Chapter 2 that can be used to provide more results in practice.

Algorithm 2 Split Testing Strategy

```
1:  procedure SplitTest(conf)                          ▷ configuration file: conf
2:      if IsTriggered(conf) then      ▷ test to determine whether payload is triggered
3:          if SizeOf(conf) = 1 then                    ▷ only one option in conf
4:              results ← GetOption(conf)               ▷ This option is pushed into result
5:          else
6:              conf1, conf2 ← Split(conf)              ▷ conf is split into conf1 and conf2
7:              SplitTest(conf1)                        ▷ perform split test on conf1
8:              SplitTest(conf2)                        ▷ perform split test on conf1
9:          end if
10:     end if
11: end procedure
```

Table 3.2: Overall Result of 34 Software Applications

Software	PLB	PEB Factors									
		FILE	REG	BIN	ORDER	MDIR	CWD	SYSDIR	ENV	CMD	All
Web Browser											
Internet Explorer	60	0	52	168	2	1	0	45	6	0	274
Opera	28	14	20	101	1	19	1	14	3	0	173
Firefox	67	189	115	284	121	5	0	38	13	0	765
Safari	99	41	265	621	1	83	2	165	10	0	1188
Google Chrome	65	2	44	177	7	34	1	64	10	0	339
PDF Reader											
Adobe Reader	46	23	22	126	0	39	6	24	5	0	245
Foxit Reader	53	0	24	125	25	0	0	30	9	0	213
MS Office											
PowerPoint 2003	35	0	35	140	1	5	0	23	3	0	207
Word 2003	39	0	40	107	0	0	0	22	2	0	171
Excel 2003	33	0	43	117	1	4	0	22	2	0	189
PowerPoint 2007	42	0	36	157	38	1	1	24	4	0	261
Word 2007	34	2	41	126	0	31	1	11	3	0	215
Excel 2007	39	0	54	132	3	0	1	21	3	0	214
Multimedia Player											
Itunes	83	46	53	138	19	2	3	59	18	0	338
Windows Media Player	60	20	19	135	6	2	0	84	1	0	267
QuickTime	69	61	66	177	14	4	0	50	24	0	396
Messenger											
Pidgin	130	11	140	298	2	160	4	25	10	0	650
Google Talk	59	2	43	139	15	0	0	42	7	0	248
Yahoo Messenger	95	0	91	202	36	3	0	63	23	0	418
Windows Live Messenger	107	24	169	257	111	5	2	43	11	0	622
Image Viewer											
Picasa 3	58	1	25	75	48	4	0	23	7	0	183
Irfan View	16	0	0	33	1	0	0	3	8	0	45
Windows Component											
Service Host	37	0	14	90	4	17	0	1	10	0	136
Accessibility Options	24	0	1	32	4	1	0	1	1	2	42
Add/Remove Program	52	0	4	95	4	34	0	2	11	9	159
Winlogon	78	0	78	160	14	67	0	4	24	0	347
SysInternal Software											
Process Monitor	34	0	8	66	8	0	0	13	8	0	103
Process Explorer	41	0	11	68	11	1	2	18	4	0	115
DLL Injection Sample											
calc.exe	20	0	1	33	4	0	0	1	1	0	40
notepad.exe	21	0	0	41	5	0	0	1	6	0	53
Other Application											
Google Earth	62	0	41	106	28	7	10	27	14	0	233
Java VM	63	55	99	254	0	102	1	40	11	21	583
IDA Pro	17	4	4	38	10	6	1	7	1	0	71
Emacs	20	2	1	40	2	2	0	4	8	0	59

3.5 Experimental Evaluation

We prototyped LDRSCOPE on an extension of TEMU [111], which is based on the QEMU virtual machine. Both the instruction trace and the system information are collected. We evaluated LDRSCOPE on 34 real-world applications in Windows XP SP3. The evaluation is intended to show the scope of complex binary loading behaviors with many non-trivial LDGs and explain the usefulness of LDRSCOPE on issues arising from binary loading, including the identification of vulnerabilities, and point out the responsible software modules which need fixing. We identified 24 binary loading vulnerabilities using LDRSCOPE. There are 21 vulnerabilities where

programs try to load binaries not present in the system, and 3 vulnerabilities where configuration files can be planted to load arbitrary binaries by the vulnerable programs. In all these cases, a remote attacker can plant either binaries or configuration files in the shared directories of the victim system and get the malicious code loading/execution. Most of these tested applications are selected in comparison with the previous approach [76], though not all of previously tested programs are selected, because of the in-availability of the older version of the software. We detected all the vulnerable loading programs the previous solution detected, with 3 vulnerabilities using configuration files, which the previous one cannot detect. We also detected and fixed a DLL injection problem where the previous solution miss to identify. Finally, we detected loading vulnerabilities through using configuration files, which they cannot detect. The static approach [78] relies on approximation without source code, which is not accurate dealing with dynamic loadings. It does not deal with the loadings through using configuration files.

Table 3.2 gives a summary of the binary loading behaviors of these applications. The rationale for this table is to provide some summary measures of the complexity of binary loading and the loading factors which are involved together with a potential vulnerability assessment. The PLB column gives the total number[8] of binaries attempted to load by each application. The columns from FILE to CMD sum up the number of dependencies for each binary, aggregating the number of loading factors for the corresponding column type. Overall, the table gives a summary of the complexity of the loading behavior per application. While the aggregate statistics is simply a rough measure, it indicates the potential attack surface of the application as every loading factor, which is not safe provides an attack chance.

We also use this summary of results to show potential vulnerability assessment. Suppose we assume the threat model contains the attacker controllable loading factors: binary search order, CWD, program command line inputs and environment variables. We can sum up the four columns to get the number of all attacker controllable factors. While the value of these controllable factors is only indicative of number of ways which a vulnerability can occur, it may be not an unreasonable way of scoring for potential vulnerabilities. Some applications are

[8]We run each application once, and count both the loaded binaries and the missing binaries that are intended to load by the application. These binaries do not include the main executable of the application or `ntdll.dll`. For each binary, we only count once.

as low as 2 (Word 2003), while others are as high as 134 (Firefox). Next we discuss several loading factors.

CWD Factor. CWD may be unsafe as it can be set to untrusted locations, i.e. shared directories. Out of the 34 applications, we identified 14 programs with the CWD loading factor. Nine of these programs attempt to load binaries missing in the system, which renders these applications vulnerable. Among the 14 applications, only Adobe Reader sets CWD to its main program directory, which can be considered safe. The others load binaries unsafely at the directory where the application associated files are opened.[9]

PEB Factors. Command line input (PEB:CMD) affects three programs – Java VM uses many command line arguments; "Accessibility Options" and "Add/Remove Program" are Windows utilities that invoke `rundll.exe` to load binaries from command line. Most programs depend on search order (PEB:ORDER), which may be a rich source of attacks. Surprisingly, Java VM does not rely on PEB:ORDER but has other issues. Note that even if ORDER is not a loading factor, CWD can still be involved, because we exclude CWD, MDIR and SYSDIR when counting the ORDER. Microsoft Word 2003 does not have ORDER and CWD as loading factors and one could speculate if the code is more carefully written given the long history of Microsoft software being targeted by attackers. Compared to PE:CMD, PE:ENV and PE:ORDER are better avenues for attackers to exploit a program, as most of the programs are affected by these two factors. Especially, PE:ENV is affecting every program. Although it is affecting more programs than PE:CMD, the effort spent in exploiting environment variables could be much higher than for command line inputs.

A summary evaluation like this could be an initial checklist for investigating whether a program is following good security with respect to its use of binaries. This can trigger a more detailed investigation of whether further problems can exist as shown in the following subsections. As many existing software may not be resistant to binary loading attacks, a tool like LDRSCOPE will be useful for software developers who we believe are not used to thinking of the security implications in the use of software modules in binaries.

In the following discussion, we introduce the use cases of LDRSCOPE. LDRSCOPE not

[9] In Windows, when a program is started by opening associated files, Windows Explorer sets an initial CWD for the program. However, the location of CWD can be changed by the program afterwards as in Adobe Reader.

only identifies the vulnerable loading behaviors, but also provides various information on the loading behaviors and loading factors that can help developers and analysts to have a better understanding to secure the binaries.

3.5.1 Loading Vulnerabilities from Configuration Files

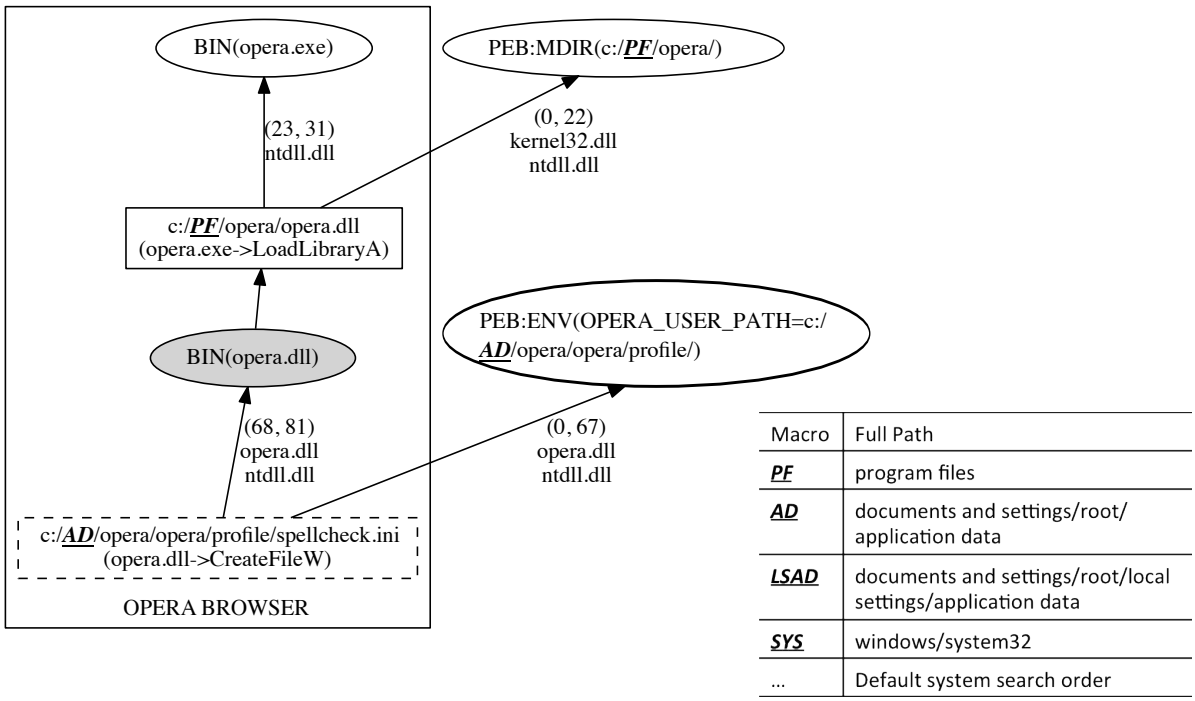
Software configuration problems can be difficult to understand and diagnose by the developers. Here, we focus on configuration files and binary loading vulnerabilities. Out of the 34 applications, 29 of them use configuration files with 27 attempts for programs to load missing configuration files. Three vulnerabilities are identified in IDA Pro, Google Chrome and the Oracle Java Virtual Machine.

First, we use the Opera browser as an example in Figure 3.6 to show how LDGs can be used to inspect the effect of configuration files on loading behaviors. The upper "LDG" in Figure 3.6 is different as the root node is a configuration file.[10] This "LDG" identifies the attempt to load a missing (presumed) configuration file `spellcheck.ini` as shown in Figure 3.6a with the dash-lined node. The intent of the configuration file is that it can be used to specify a spell checking engine component with the option: `Spell check engine`. We found that an optional configuration file with the same name can override this setting. LDRSCOPE tests this configuration file with the setting: `Spell check engine=d:/exploit.dll`. The result of this run is the LDG in Figure 3.6b. It identifies the loading factor of binary `exploit.dll` as the configuration file `spellcheck.ini` and highlights the PEB:ENV loading factor as conditionally safe based on the chosen safety constraints. A warning is generated to highlight the PEB:ENV loading factor, if an attacker can place a crafted configuration in this directory.

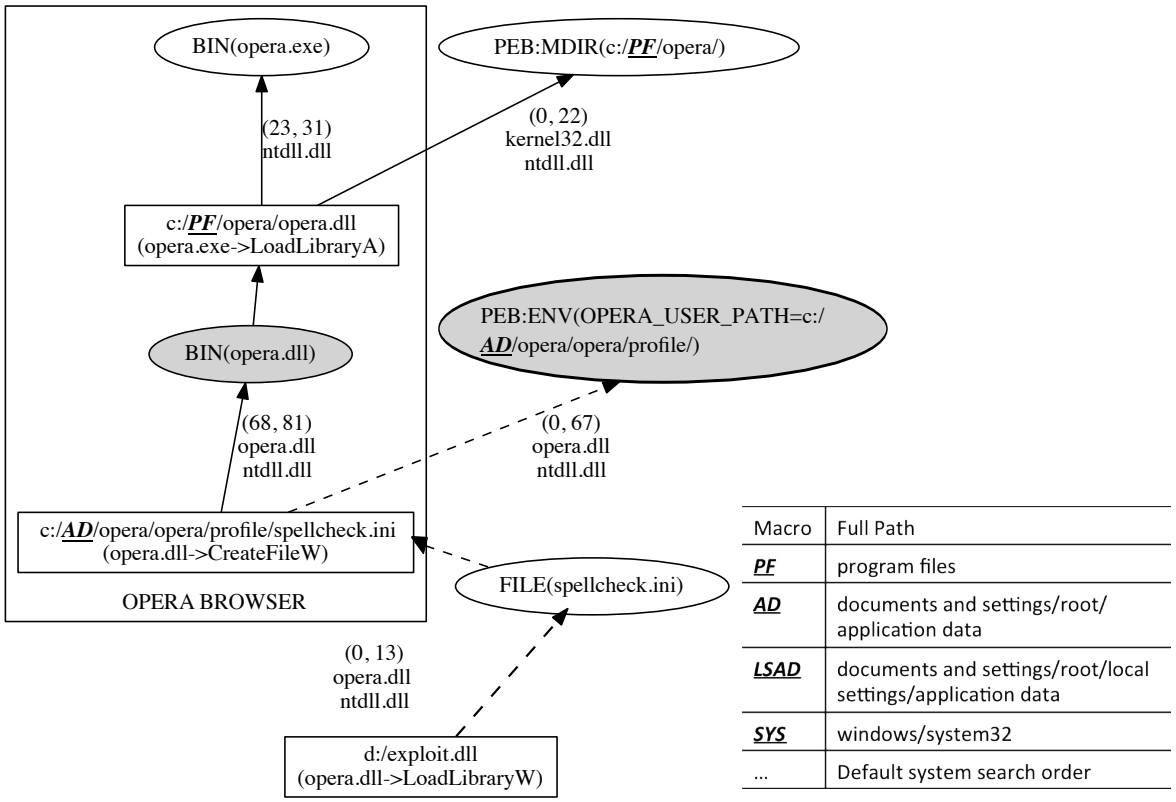
Table 3.3 shows experiments on Google Chrome, Java Virtual Machine, IDA Pro and Opera, with the configuration file options found to load binaries. We now summarize these experiments, highlighting the vulnerable loading factor.

- Google Chrome can load various binaries specified by an optional configuration file

[10] LDRSCOPE can generate extended LDGs for any file and not just binaries, extending loading factor to deal with arbitrary files.

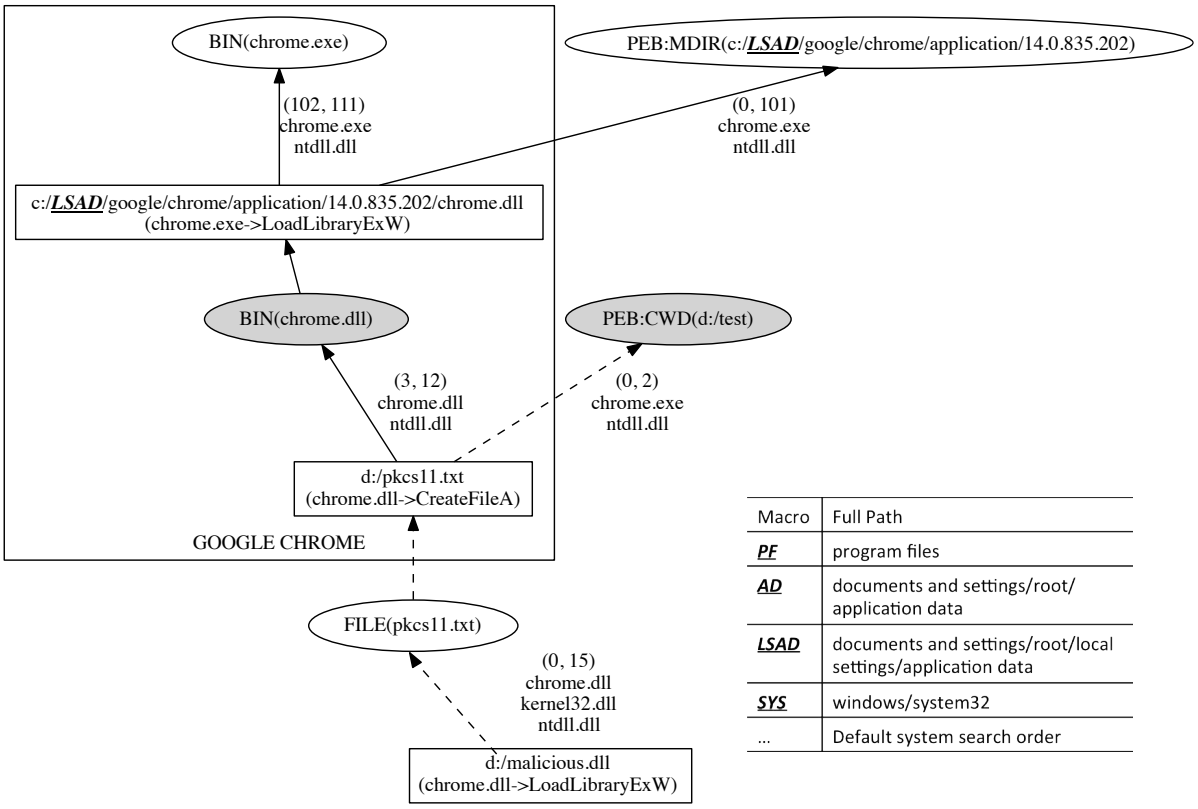


(a) Dependencies of Configuration File Loading Attempt

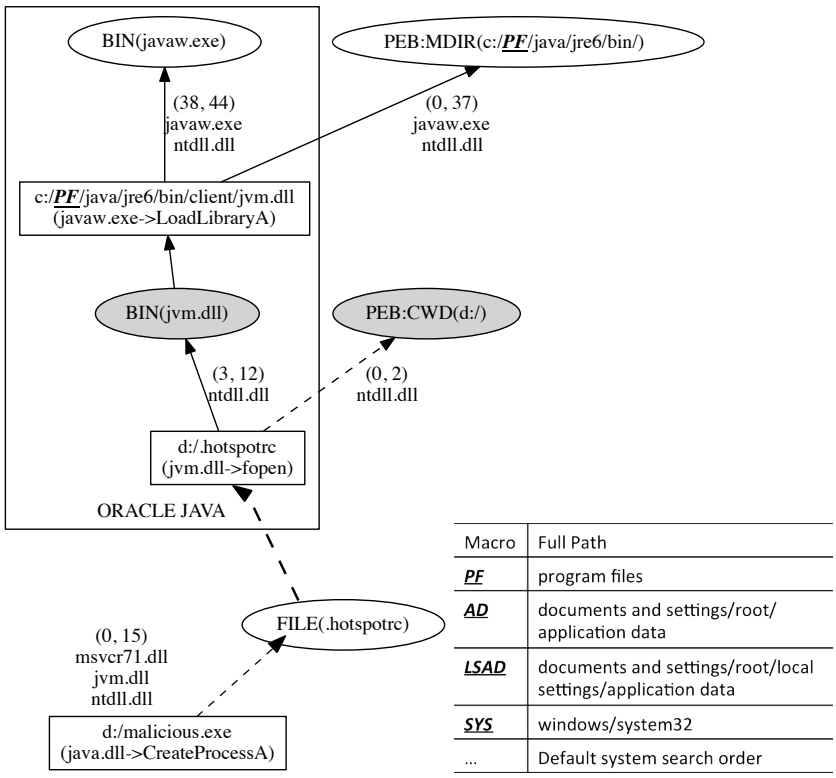


(b) LDG for Opera

Figure 3.6: Loading Behaviors Caused by Configuration Files



(a) LDG for Google Chrome Vulnerability



(b) LDG for Java Virtual Machine Vulnerability

Figure 3.7: Loading Vulnerability Caused by Configuration File in Unsafe Locations

Table 3.3: Loading Behavior Caused by Configuration Files

Software	Configuration File	Option To Load Binaries
Google Chrome 14	pkcs11.txt	library
Java VM 6.26	.hotspotrc	OnOutofMemoryError
IDA Pro 5.5	idauser.cfg	GRAPH_VISUALIZER
Opera 9.64	spellcheck.ini	Spell check engine

`pkcs11.txt`. When an `https` request is first issued in the browser, Google Chrome searches in the root directory of CWD for this file. The setting `library=d:/malicious.dll` in the optional configuration file causes Chrome to load `malicious.dll`. It turns out that this is a known vulnerability in the Mozilla Network Security Services (NSS) library, which is statically linked into `chrome.dll`. Figure 3.7a shows the loading dependency from the payload `malicious.dll` to the configuration file, marked by the dash-lined path. The configuration file is located at the root directory (`d:/`) of CWD (`d:/test`). This location, which LDRSCOPE highlights, is a shared directory in the evaluated Window system, which can be accessed by the attackers to plant malicious binaries. The LDG shows the binary `chrome.dll` load both the configuration file and the malicious payload binary.

- The Oracle Java Virtual Machine (JVM SE 6u26) loads and executes an error handler binary specified by the option `OnOutOfMemoryError` in the configuration file `.hotspotrc` from CWD. To exploit this vulnerability, one can set the option `OnOutOfMemoryError=d:/malicious.exe`. The configuration file is read when JVM starts. The error handler is loaded when the heap runs out of memory. The LDG in Figure 3.7b shows `malicious.exe` depends on the configuration file `.hotspotrc` with CWD as the loading factor. LDRSCOPE highlights the path to the unsafe loading factor showing CWD and the binary `jvm.dll` used to load the configuration file. The binary `java.dll` invokes `CreateProcessA` to load the malicious payload. The problem is in the Oracle Java's components.

- The IDA Pro configuration vulnerability discussed in Section 3.2.1 is given in Figure 3.3. It is interesting that the IDA Pro documentation in `ida.cfg` suggests that the user configuration file `idauser.cfg` must be placed in a safe location, such as `/IDA/cfg/`.

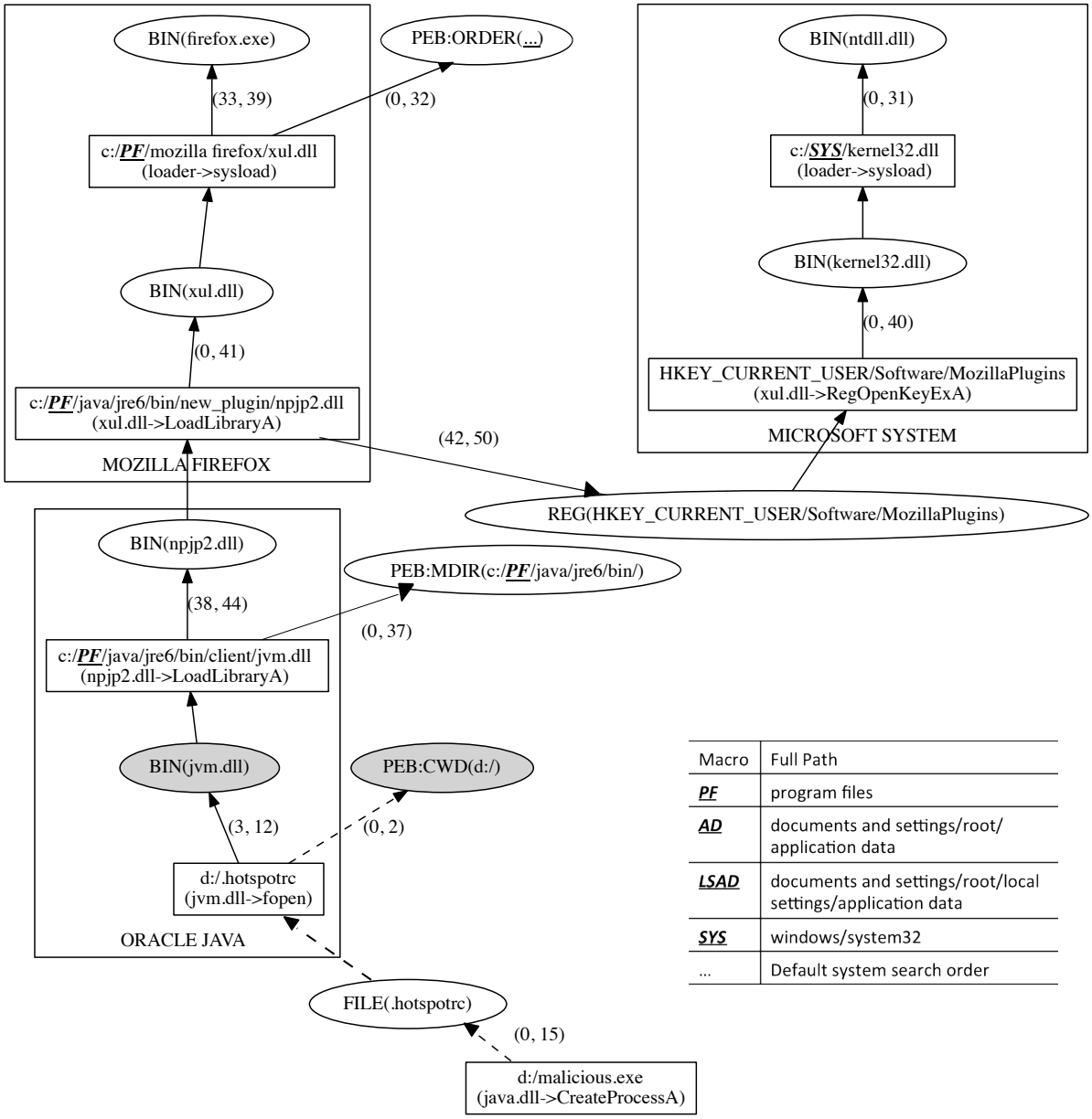
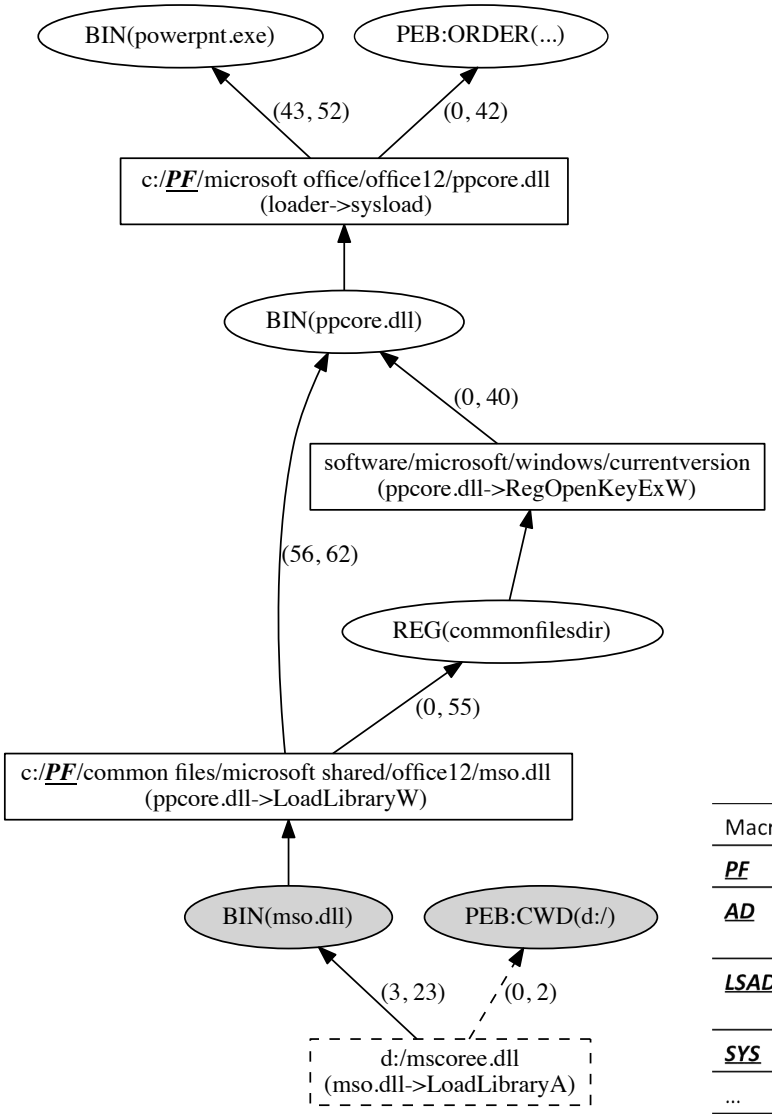


Figure 3.8: Abridged LDG for firefox vulnerability

Our analysis shows that the IDA Pro code does not check for safety as it still attempts to load `idauser.cfg` from an unsafe CWD directory.

Besides the above cases, we also found many programs (Google Earth, Pidgin, Yahoo Messenger, Firefox, etc.) attempt to load missing configuration files (`search.ini`, `override.ini`, `ymsgr.ini`, `secmod.db`, etc.) with only a relative path. Such behavior can be susceptible to file planting attacks [5].

Our evaluation shows that weaknesses in how software works with configuration files can be quite common. Finding and fixing configuration vulnerabilities can be quite challenging.



Macro	Full Path
PF	program files
AD	documents and settings/root/ application data
LSAD	documents and settings/root/local settings/application data
SYS	windows/system32
...	Default system search order

Figure 3.9: Abridged LDG for PowerPoint 2007

LDRSCOPE can highlight to developers that they need to pay attention to not only loading behaviors but also how they handle configuration files.

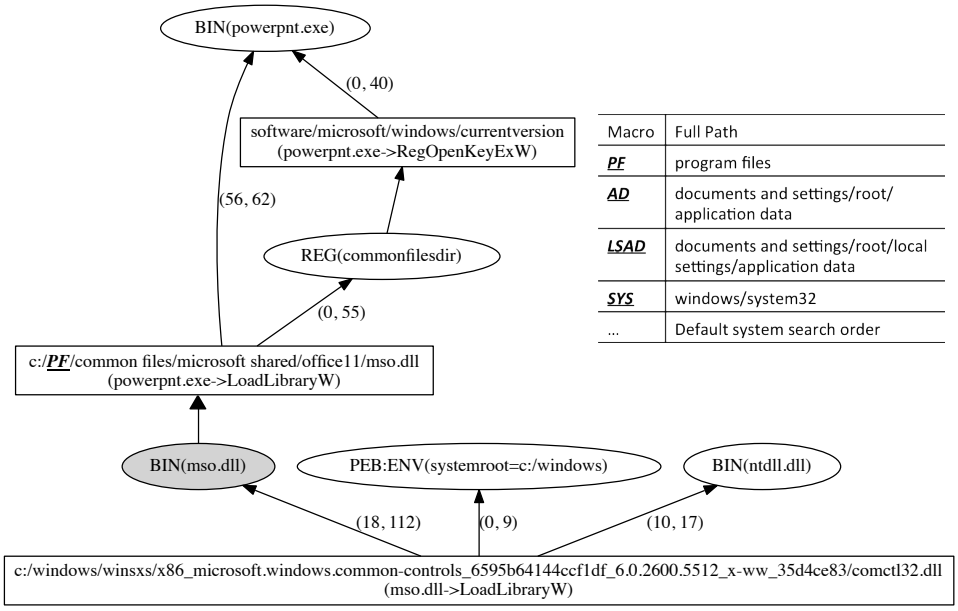
3.5.2 Safety of Third Party Binaries

Programs using third party libraries can be vulnerable due to the defects in these libraries rather than in their own code. This is significant because a binary loading vulnerability outside the application code is not easily remedied, and the developers only control their own code, not the libraries from other vendors. Programs can be vulnerable due to the use of vulnerable components or plug-ins. LDRSCOPE can provide provenance analysis on binary loading behaviors with LDGs.

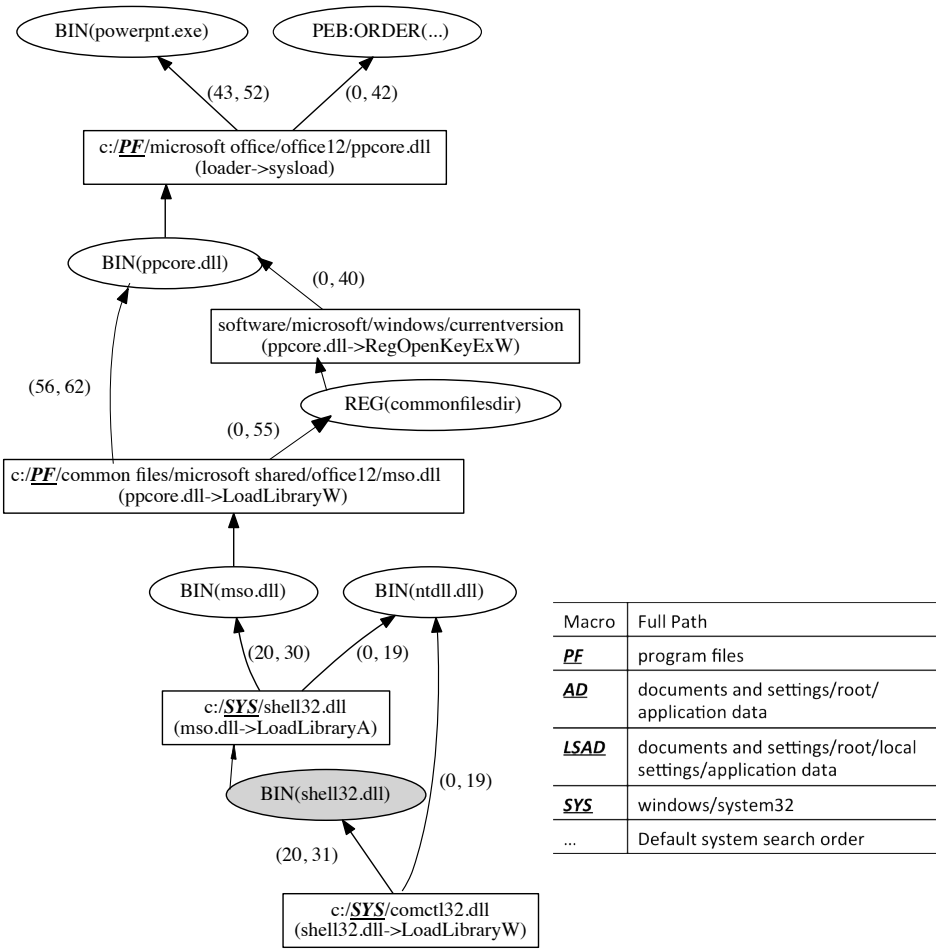
We found that Firefox becomes vulnerable due to the use of a vulnerable JVM plug-in, as shown in Figure 3.8. In order to support Java applets, Firefox loads `jvm.dll`. However `jvm.dll` contains the vulnerability shown in Figure 3.7b. The third-party DLL, `jvm.dll` causes the problem (details discussed in the previous section), which is not in the code base of Mozilla Firefox. Figure 3.8 shows the grouping of binaries involved in the loading behavior of `malicious.exe`, the binary causing the problem is grouped to *Oracle Java*, which clearly shows the provenance of the problem. Thus, not only do we detect vulnerable loading factors, we also identify whether the source of the problem is in the application code or due to third-party code.

3.5.3 Loading Vulnerabilities from Missing Binaries

Binaries that are missing can lead to vulnerabilities if the missing binary can be controlled by the attacker. LDRSCOPE detects 21 loading vulnerabilities from 12 programs, which attempt to load a binary that is not present in the system, including Google Earth, Apple Safari, Apple Itunes, Windows Live Messenger, Yahoo Messenger, Pidgin, Office Word 2003, Office PowerPoint 2007, Excel 2007, Word 2007, Process Explorer and Picasa. A typical LDG for this kind of vulnerability is shown by Figure 3.9, where the dash-lined node indicates the missing binary that PowerPoint 2007 attempts to load. Since `mscoree.dll` is missing, the program looks for the binary according to the search order, and ultimately will end up looking in CWD with



(a) Abridged LDG for PowerPoint 2003



(b) Abridged LDG for PowerPoint 2007

Figure 3.10: Comparison of Loading Behaviors

the default search order. LDRSCOPE not only detects the loading behaviors that happened, but also captures the attempts to load binaries. In this way, the binary loading vulnerabilities from missing binaries can be identified.

Figure 3.9 shows a DLL `ppcore.dll`, whose loading factor is the binary search order. There are 36 loading behaviors, which directly or indirectly depend on this DLL. Since this binary depends on the search order, which may be unsafe, all the loading behaviors relying on this binary can become vulnerable. We can think of this binary as a "core binary" because many other binaries depend on it. Therefore, software needs to ensure the safety of core binaries, since they can affect many other binaries.

Examining Figure 3.9 turns up another discovery on binary sharing. In the figure, `mso.dll` attempts to load the missing binary `mscoree.dll`, which leads to the vulnerability discussed above. It turns out that `mso.dll` is shared across all the major Office components in the system, which propagates the vulnerability to all the programs using it. This means that besides PowerPoint, Word, Excel and other Office components all have the same vulnerabilities due to the sharing of the binary `mso.dll`. So finding a binary loading bug due to a binary in one software will likely lead to also finding the same vulnerability in other software if the binary is a binary common to all the software. Leveraging bug discovery across a suite of software leads to reduced effort in bug fixing. The LDGs generated by LDRSCOPE can be used together to provide statistics on loading behaviors and bring up new discoveries across different software components.

3.5.4 Comparison of Loading Behaviors

LDGs can be used to study the evolution of loading behaviors across different versions of a software. Figure 3.10 shows the loading of the binary `comctl32.dll` in both PowerPoint 2003 and 2007 on a Windows XP SP3 system. In Figure 3.10b, PowerPoint 2007 first loads the binary `mso.dll`, which then loads `shell32.dll`. `comctl32.dll` is loaded due to the loader `ntdll.dll` and `shell.dll32`. In Figure 3.10a, PowerPoint 2003 also loads `mso.dll`. However, `comctl32.dll` is loaded due to the loader `ntdll.dll`, an environment variable `systemroot` and this `mso.dll`.

The different loading factors in this example illustrate a Windows feature, side-by-side assembly [6], which was introduced to allow multiple versions of binaries to exist in the system. For example, it allows an updated system to be still compatible with older versions of software, which also depend on older binaries. Although the two different version of PowerPoint use `comctl32.dll` with the same name, the versions of the DLL are different. PowerPoint 2003 needs an older version of `comctl32.dll`. The location of this older version is in the side-by-side assembly cache shown by the pathname in Figure 3.10a. Use of side-by-side assembly can introduce more loading factors, which have to be checked if they are vulnerable or not. In this example, the `c:/windows` directory is safe. However supporting more versions of a binary may also increase the number of attack vectors. LDRSCOPE alerts developers to issues with such features.

3.5.5 Inspection of Binary Injection

Binaries can be injected into target program by other processes. Such injections are considered as a special kind of binary loading behavior. These behaviors are common for programs that want to modify the behavior of other programs, e.g. TortoiseSVN injects its DLLs into the Windows Explorer shell process to enhance the user experience, anti-virus software usually injects binaries to other programs to function properly, etc.

We observed that previous dynamic approaches [76, 77] do not give a clear explanation for injected binaries. For example, Google Desktop injects `googledesktopcommon.dll` to multiple programs, but as their hooking techniques do not deal with the case of injected binaries, an accurate explanation is not given. LDRSCOPE tries to detect all loading behaviors and be complete in the explanation of binary loading with respect to the process being studied. In Windows, there are a number of DLL injection techniques [7], such as using the registry key `AppInit_Dlls` and using API call `CreateRemoteThread`. Table 3.2 gives a summary of our binary injection using both these techniques on `Calculator` and `Notepad` in the DLL injection sample row. LDRSCOPE accurately attributes the different injection techniques to registry (REG) and untracked (UNK) loading factors.

When inspecting the LDGs of `Calculator`, we find that the loading of an injected binary

is due to the loading factor REG, a registry key `AppInit_Dlls` set as an absolute pathname of the injected binary. The LDGs can be used to identify the DLL injection technique using registry keys. In another experiment, we use an injector program to deliberately inject a binary into the `Notepad` program by invoking the `CreateRemoteThread` API and a series of other APIs. When inspecting the LDGs, we find the loading of the injected binary is due to an untracked loading factor. This factor shows an address out of the memory space of `Notepad`, but in the memory space of the injector program. Thus, the loading factor is UNK.

3.6 Related Work

A number of research works have been proposed to identify unsafe binary loading behaviors. Dynamic approaches in [76, 77] are similar to ours in that they analyze the trace from program instrumentation/monitoring to identify unsafe dynamic component loading. However they work by detecting *resolution failure* and *resolution hijacking*, which only addresses a subset of binary loading vulnerabilities - mainly those to do with the path resolution process and do not explain why the loading happens. For example, they fail to detect unsafe loading behaviors specified by using an absolute path. In their approach, the loading behavior in the IDA PRO example would be considered as safe. An alternative is static analysis, [78] analyzes the call sites of loading APIs and computes the execution slices to figure out the possible loading behaviors. The slices are emulated to guess the possible binaries that could be loaded. Unfortunately, due to the dynamic features of loading behaviors and the lack of source code in binary, it cannot analyze sophisticated loading behaviors, e.g. the ones caused by configuration files, which cannot be determined in advance using static analysis. We differ from these works in that we focus on understanding the binary loading behavior in a detailed way and producing a comprehensive explanation in terms of loading factors which is obtained through dynamic tracing.

In order to extract loading dependencies, we employ dynamic dependency analysis techniques. Dynamic tainting analysis [94] provides forward data dependency analysis using taint propagation. However, forward dependency analysis makes it more difficult to identify the

loading dependencies than the backward dependency analysis we employ. LDRSCOPE utilizes backward data dependency analysis similar to [26] on the loading behaviors. It also employs system dependency analysis to avoid the overhead and complexity of tracing the whole operating system. System dependency graphs are used for slicing the program [83, 109] and application debugging [105]. These graphs focus on the programs, while our system dependency analysis brings in OS level objects.

Protection mechanisms are proposed to enhance the binary safety. Safe Loading [101] prevents programs from code-oriented exploits, which utilizes the combination of system call policy and control flow integrity. It takes the application and related libraries into a sandbox, and applies translation of the application code. However, it does not prevent the binary loading attacks, because they do not violate control flow integrity. BinInt [124] provides binary integrity by restricting the execution of software. It limits the binary installation through allowed channels. It prevents external binaries from executing. However, this security model does not prevent binary loading attacks leveraging the existing binaries already installed.

Orthogonal approaches can be applied to enhance the exploratory testing for configuration files and their options. Tupni [46] is a tool to reverse-engineer an input format with a set of information based on the observation that applications usually process iterative data records in loops. It can identify record sequences and types in input data, and even detects different types of constraints on the values. We apply similar ideas to locate the configuration file parsing logic and test for configuration options. EnCore [130] applies data mining techniques to detect software misconfigurations. It considers two factors the interactions between the configuration settings and the executing environment, and the correlations between the configuration entries. This approach can help detecting configuration options but it also requires configuration file datasets to get correct results.

3.7 Summary

In this chapter, we develop an approach for accurately understanding binary loading behaviors. We focus on Windows where most of the binary loading problems and vulnerabilities

occur. Our analysis combines data dependency analysis with system dependency knowledge to identify the factors affecting the loading behaviors of the program. The end result is a loading dependency graph, which explains binary loading by showing the dependencies from loading behaviors to the factors. We evaluated the prototype of our solution on 34 Windows applications, and found 24 binary loading vulnerabilities; three of them are caused by using configuration files in an insecure fashion. Our approach precisely identifies the binaries that cause the vulnerability.

The main motivation of LDRSCOPE is how to help software developers deal with the complicated binary loading behavior involving their code, Windows software components and the Windows operating system kernel, multiple third party binaries, configuration and data files, etc. The LDG simplifies these interactions and complexity with an understandable explanation. We demonstrate several use cases of how to use LDGs to build secure programs.

For the test case generation, we plan to make the testing of configuration file "smarter". We can create heuristics from various related analysis [46, 130] to find configuration options controlling the binary loading behaviors. Currently, the test cases contain the initialization of programs. We can add more user interactions to the tested programs, so that the dynamic loading behaviors can be traced, if they can be activated in the interactions. For the program tracing mechanism, we plan to reduce the size of the traces in the future, as the size of a trace can be more than 50 gigabytes for a program to initialize itself.

Chapter 4

Detecting API Misuse Vulnerabilities

4.1 Introduction

In this chapter, we introduce a problem where component interactions in a complex computer system can affect the behaviors of software components. In this case, browser components and browser plugins interact with each other. We propose a solution to detect the interactions that can allow a component to access the system privilege it is not supposed to access. We also secure the interaction between these components in a fine-grained manner.

Software systems often employ reusable components. These components often have different privileges or permissions. Some components are written to be general-purpose and not specifically designed for a particular program or application, thus often provide more functionality than what is needed by the applications using them, as well as having more privileges than an application needs.

When a program is benign, the added functionality and privileges from software components are not used. However, when the program is exploited through a vulnerability or is malicious, the additional functionality and privileges are activated by the interactions among the components. Thus, the privilege[1] carried by the component is effectively re-delegated to the program through the misuse caused by component interactions, forming an attack. For example, JavaScript in a web browser is normally restricted from accessing files in the oper-

[1]The privilege indicates the functionality designed in the software component, but is not meant to be used in certain environment.

ating system by the browser sandbox. In this sandbox, JavaScript is only allowed to perform web-related actions, instead of the general-purpose programming tasks, e.g. writing files, creating processes. However, in the Microsoft Internet Explorer (IE) browser, ActiveX controls[2] can be leveraged by malicious web pages to allow JavaScript to access files in the local file system. Due to the complexity of the Windows system, even if users are fully aware of the functionality of such ActiveX controls, they may not be able to detect the threats from the ActiveX controls when component interactions are involved. Software components can expose their functionalities unexpectedly to other components through misuses caused by component interactions. We call such a vulnerability, an *API misuse vulnerability*, which is the focus of this chapter: the privileges of a software component, e.g. an ActiveX control, are exposed to another software component, through the unexpected use fashion of the component in component interactions. The misuse in an ActiveX control is mainly for gaining privileges. The abuse of APIs in an arbitrary fashion is not the focus of this chapter. API-misuse vulnerabilities can occur in many different forms, where software components are interacting with each other to activate the hidden functionalities in them. Some examples are: (i) API misuse problems in ActiveX components (the main focus of this chapter); and (ii) permission re-delegation problems in Android components [57] (to be discussed in Chapter 5).

In this chapter, we introduce a solution to detect API misuse vulnerabilities in software components, focusing on the problem in the context of ActiveX controls in Windows. Our approach introduces a privilege reachability model (PRM), which describes the interactions of different software components in a form of access path. The API misuse vulnerabilities are detected by analyzing the access paths in this model. We also propose a prevention mechanism to effectively block the attacks leveraging this kind of vulnerabilities in Internet Explorer (IE).

For the rest of the chapter, in Section 4.2 we introduce ActiveX together with a motivating example of the API misuse vulnerability. We present the detection mechanism of API misuse vulnerabilities in Section 4.3. We present a prototype implementation tailored for ActiveX controls in Section 4.4 and describe a mitigation solution. Section 4.5 gives the discussion of experimental evaluation results. Section 4.6 discusses the related research works, and Sec-

[2]A detailed example is in Section 4.2

tion 4.7 concludes the whole chapter.

4.2 Problem Definition

In this section, we describe a typical example of an API misuse vulnerability in an ActiveX control, and define the problem addressed in the whole chapter.

4.2.1 A Motivating Example

Background of ActiveX. ActiveX controls [8] are a mechanism to build reusable software components on the Microsoft Windows platform. It is widely used by many Windows applications, including Microsoft Office, Windows Media Player, and Internet Explorer (IE), allowing applications to use the functionality embedded in ActiveX controls. IE allows methods in ActiveX controls to be accessed from web pages, through using the JavaScript engine. ActiveX controls in the browser program are native binaries, thus giving web pages the ability to run native code in the operating system, e.g. Windows Update ActiveX Control in Windows XP and Visual Studio (Windows 7) use IE to apply Microsoft updates.

In order to use an ActiveX control in a program, users first install and register it in Windows, e.g. install an ActiveX control as part of a software through the standard package installer. In addition, programs like IE can load remote ActiveX controls. Once an ActiveX is available, the program hosting it can create ActiveX objects, through which the program can invoke methods in the ActiveX controls. ActiveX methods are exported functions to other applications, which are similar to the member functions in C++. IE also allows JavaScript and VBScript in a web page to create ActiveX objects and invoke ActiveX methods in the *scriptable interfaces*. Figure 4.1 shows an example code snippet to create an ActiveX object and invokes three methods of the Office Snapshot Viewer ActiveX Control using JavaScript in IE.

Existing security mechanisms for ActiveX rely on the trust from end users. Take IE as an example, only trusted ActiveX controls can be accessed by web pages. Local ActiveX controls are all trusted, except those blocked by compatibility flags in the registry (*killbits* [9]) by default. For remote ActiveX controls, users can customize a white list of trusted sites [10]

```
1 <script language='JavaScript'>
2     //Create Activex Object with ProgID
3     var obj = new ActiveXObject("snpvw.Snapshot Viewer Control.1");
4     // invoke method SnapshotPath, CompressedPath, ...
5     obj.SnapshotPath = "c:\\TestSnapshot.snp";
6     obj.CompressedPath = "c:\\TestSnapshot-compressed.snp";
7     obj.PrintSnapshot("True");
8 </script>
```

Figure 4.1: Using Microsoft Office Snapshot Viewer ActiveX Control in JavaScript

```
1 <script language='JavaScript'>
2     //Create Activex Object with ProgID
3     var obj = new ActiveXObject("snpvw.Snapshot Viewer Control.1");
4     //download payload
5     obj.SnapshotPath = "http://malicious.com/payload.exe";
6     //save to local file system
7     obj.CompressedPath = "c:\\clickme.exe";
8     obj.PrintSnapshot("True");
9 </script>
```

Figure 4.2: An Attack on Microsoft Office Snapshot Viewer ActiveX Control in JavaScript

to permit remote ActiveX control usage within the sites. Otherwise, IE prompts the user for permission to load the control, but users usually don't know whether the ActiveX control is safe or not. In order to use the webpage embedding the control, they tend to allow the controls. In addition, IE only initializes and utilizes the ActiveX interfaces in certain settings, where the *Safe for Initialization* and *Safe for Scripting* properties are implemented – this also requires trusting the control from users.

An ActiveX Attack Example. As a native shared component running in a low privileged scripting environment, the ActiveX control can expose higher privileged native access to system resources to the software component interacting with it. We show the API misuse problem with a real-world example. The Microsoft Office Snapshot Viewer ActiveX Control is a shared component of Microsoft Office Access 2003. This ActiveX control is designed to print and view snapshot of documents in the web browsers. Figure 4.1 shows a typical usage of the control. The `SnapshotPath` and `CompressedPath` methods specify the paths of the snapshot file and the compressed version of the file to be displayed in the Snapshot Viewer. `PrintSnapshot` prints the snapshot file on the screen.

In IE, browser scripts, such as JavaScript, stay in the sandbox of the browser. The sand-

box allows only the web related actions, such as open links and view images, but does not allows operations, such as native access to the operating system. However, by leveraging on the ActiveX control, an attacker can bypass the restrictions of the IE sandbox, and allow the JavaScript to access local file system. A malicious JavaScript program as shown in Figure 4.2 can invoke `SnapshotPath` with `http://malicious.com/payload.exe` to silently download the payload and invoke `CompressedPath` with `c:/clickme.exe` to save the malicious payload to the local file system. This is a drive-by download attack caused by the shared component with access to privileged APIs in the ActiveX control. These privileged APIs grant the JavaScript with additional native functionalities, which are normally not allowed.

Such API misuse vulnerabilities are achieved through the unintended exposure of native privileged operations in software components. One component does not affect the system by itself; instead, it tries to gain more privilege through interacting with vulnerable software component with higher privileges. The interactions cause the misuse of privileged APIs in these components. Hence, we need a systematic approach to detect API misuse vulnerabilities in shared software components, such as ActiveX controls.

4.2.2 The Privilege Escalation Problem in Component Interactions

First, we define the abstract problem of privilege escalation as follows. Assume there are two software components that have different privileges: component $C1$ with the privilege set $P1$, component $C2$ with the privilege set $P2$. $C1$, the *client component*, can request service from $C2$, the *service component*, through $C2$'s interface. If through $C2$, $C1$ can gain privileges not in $P1$, we call this a *privilege escalation* in component $C2$.

The cause of the problem is that some software components expose privileged access of system resources through certain API or interface during component interactions. Without loss of generality, we consider only two privileges, low (client component) and high (service component), and it is easy to extend this to more privileges or hierarchies.

In the context of using APIs from a program, as with "local attacks" discussed above, the usual objective is to gain more privileges. Thus, we focus on misuse which aims at some form of privilege escalation. In this context, we see that the API misuse vulnerability is one instance

of the privilege escalation problem during component interactions. The privilege escalation happens because the software components are interacting with each other, so that the hidden functionalities are activated by the interactions.

In the ActiveX example, the JavaScript engine component in a web browser program is a client component with its input being JavaScript code in a web page. The ActiveX controls are service components. The privileged APIs, such as saving file to local storage described in the example, can be accessed through a vulnerable ActiveX control, causing escalated privileges in the JavaScript engine component.

This vulnerability is not only restricted to ActiveX controls, software components, such as Android components, can also have the same problem. One example is in the permission re-delegation problem [57]. This specific Android component interaction feature can be leveraged by attackers to initiate privilege escalation attacks.

In this chapter, we focus on the detection and prevention of this problem by identifying API misuse vulnerability in ActiveX controls and blocking the accesses to privileged APIs. For the Android privilege escalation problems, we treat them in Chapter 5.

We discuss the API misuse problem in ActiveX controls in Windows with following aspects:

- **No Source Code.** Due to the closed source feature of Windows system, binary components such as ActiveX controls do not come with any additional information, e.g. symbol tables. Dynamic analysis is more effective to be applied to the API misuse problem than static analysis for two reasons. First, we are dealing with pure binaries in Windows without source code. Binaries also pose many more difficult problems for static analysis. Complete disassembly for static analysis is an undecidable problem [120]. For example, IDA Pro (which is considered by many to be the best disassembler still in practice) still suffers from errors [120]. Second, we are dealing with dynamic behaviors in the ActiveX controls, which are hard to analyze accurately by static analysis. Moreover, the size of Windows programs, which can have a code base of millions lines of code, makes it even harder to apply static analysis. Dynamic analysis is more accurate but incomplete. We make a trade-off to use dynamic analysis in our solution.

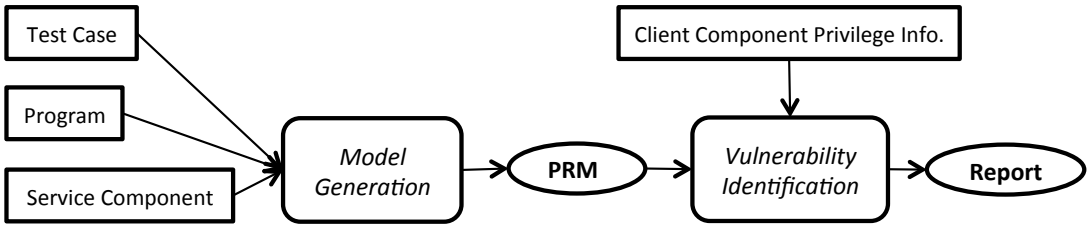


Figure 4.3: Overview of API Misuse Vulnerability Detection

- **General Testing.** Test cases are usually used in dynamic instrumentation. Testing strategies and heuristics are orthogonal to our approach, in that they can provide intensive and comprehensive use cases of the ActiveX behaviors that are analyzed. As a general ActiveX analysis framework, our system accepts user defined test cases and existing use examples of the ActiveX controls. If none of these test cases is available, our system can automatically generate default test cases invoking the ActiveX methods. This benefits from the *type libraries*[3] in ActiveX controls, where the argument types of the ActiveX methods can be extracted. Knowing the argument types, we can fill in the arguments according to certain heuristics.

4.3 API Misuse Vulnerability Detection

We now describe an approach for detecting API misuse vulnerabilities, with an overview given in Figure 4.3. Our approach analyzes a service component to extract a model describing the reachability of its privileges. Combined with the client components interfaces and privilege settings, our approach detects the presence of an API misuse vulnerability in the service component.

The whole approach consists of two main phases: privilege reachability model generation and vulnerability identification. To decide the reachability of privileges of the service component, we test the service component in a hosting program driven by a set of test cases. It may be case that we have to find the privileges of the software components as it may not be explicitly given. To determine reachability, we consider the caller-callee relationship along an execution path. The paths found can be used to generate a (execution) profile in the *model*

[3]A binary file stores information of the ActiveX object's property and methods that is accessible to other applications.

generation phase. From the profile of the service component, our approach builds an abstract model to represent the reachability between the component interfaces and its privilege, which we call the *privilege reachability model* (PRM). Combining this with knowledge about the client component privilege, in the *vulnerability identification* phase, we find potential API misuse vulnerabilities. In the rest of this section, we give more details of this steps focusing on API misuse vulnerabilities in ActiveX controls.

4.3.1 Dynamic Instrumentation

We track the control flow and collect execution paths by using dynamic instrumentation based on the Intel PIN framework [86]. We create a tracking tool to record the control flow of a process running in Windows. Tracking control flow between functions is less straightforward than it might seem because we need to handle non-local control flow, e.g. setjmp/longjmp, exceptions, kernel callbacks and multi-threaded execution. We found no difficulties with running the IE browser after the dynamic instrumentation using the tool.

We use test suites, which may come from existing test code of the ActiveX controls or manually written test programs, to carry out the dynamic instrumentation tests. We additionally develop a test template generator, which automatically generates template test cases that directly invoking the exported APIs in ActiveX controls. The generator extracts all methods in the *IDispatch* interfaces in the ActiveX controls according to the type libraries, and creates HTML webpages embedding the appropriate JavaScript code to invoke the extracted ActiveX methods. The ActiveX methods are invoked using fixed values for their argument types stored in the type libraries. The test cases generated this way are templates, since the arguments of the methods are filled in by applying heuristics, e.g. using specific values based on their types. Manual assistance can be applied to adjust these arguments to make the template functioning like real use cases. Fuzzing techniques [60, 114, 127] can be incorporated to explore the corner cases in the usage of ActiveX control. However, the focus of API misuse detection is on the normal usage of APIs. Even with the basic test cases, we can already find many vulnerabilities in ActiveX controls. We expect better results and coverage with in-depth testing in the future.

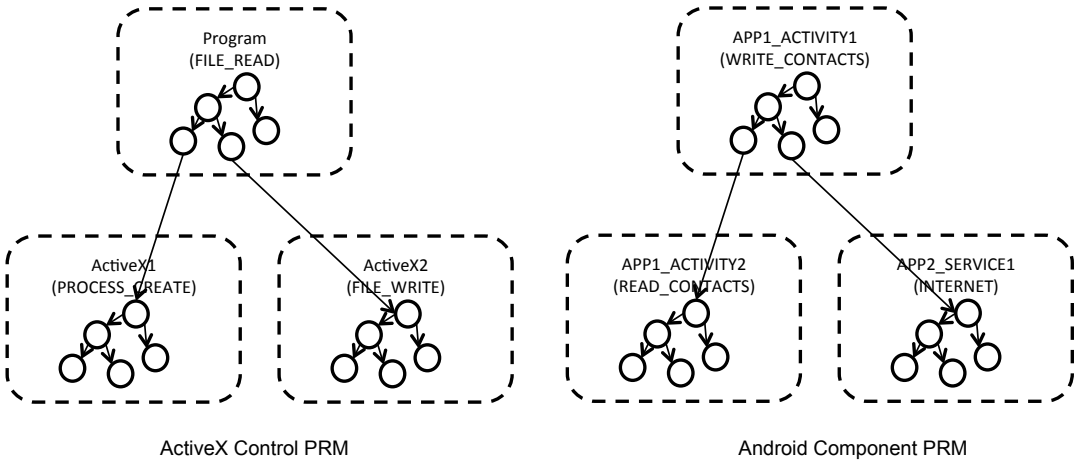


Figure 4.4: Privilege Reachability Models for the Shared Components

4.3.2 Privilege Reachability Model Representation and Generation

Model Representation. The privilege reachability model employs a graph representation, analogous to a call graph. Note that such a call graph may not be readily available even in the (hidden) source code due to the indirection and the dynamic feature of Windows components. Every node represents a function invoked during the execution. Every directed edge represents the invocation from a caller to a callee function. Nodes belonging to the same software component are grouped together. The group represents the shared components and shares a same privilege. The privilege is predefined by specification of the software components. We give two examples. ActiveX control's privileges are defined by the system calls it can access to achieve certain privileged operations in the system, e.g. write a file. The privileges of Android components are the Android permissions defined in the manifest files. As shown in Figure 4.4, the ActiveX control model contains a program with FILE_READ (read files) privilege interacting with two ActiveX controls with PROCESS_CREATE (create processes) and FILE_WRITE (write files) privileges. Similarly, the Android component model contains two Android Activities with permissions WRITE_CONTACTS and READ_CONTACTS, and one Android Service with permission INTERNET.

In order to describe the reachability in the PRM, we introduce the notion of *access path* and define various distinguished functions. We start by introducing the functions first, and then we show how the access path forms from these functions. *Privileged APIs* are system APIs that provide privileged access to system resources. For example, the `NtWriteFile` API in

Windows is used to write files in the system, thus, it can be thought of as an interface having file write privileges. *Entry functions* are exported functions in the shared components, which are the interfaces used by clients and also other services. An *access path* is a path in the graph of the model from a low privilege component to an entry function of a high privilege component to a privileged API. Thus, an API misuse vulnerability is an access path in the PRM to the high privilege component.

Model Generation. To build a PRM, our approach first collects the execution paths from the interactions between a program and associated service components. There are two typical approaches for path collection. One is static analysis. However, many of the APIs make heavy use of indirection and callbacks, which pose difficulties for static analysis. The other choice is dynamic analysis, where we can collect execution path information through dynamic monitoring during program execution. Execution traces of the software components give the control flow in terms of function calls. We assume that there is knowledge about the service components and the availability of test cases for performing normal functionalities of the program component. Such test cases can be obtained from software examples or the software documentation.

The model generation step takes the execution paths as the inputs and builds the PRM described above. For the ActiveX control, it first recovers the call sequences from the real execution traces. Then, it groups the functions from the same shared component. Finally, it adds metadata to the graph, such as, labels showing privileges of the shared component. This step also supports generating compressed graphs for visualizing the relationship of the function calls. More details are discussed in Section 4.4.1.

ActiveX PRM. We describe how the above model specifications are applied on the ActiveX controls. There are four types of nodes in a PRM of an ActiveX control:

- *ActiveX methods* are entry points of ActiveX controls, which are exported APIs from the scriptable interfaces, e.g. the `SnapshotPath` method in the motivating example.
- *ActiveX inner functions* are functions in the ActiveX control that are not exported. Since we extract them using the ActiveX binaries, they are usually denoted by their addresses

in the binaries. For example, `snapview.c5ad` is such a function.

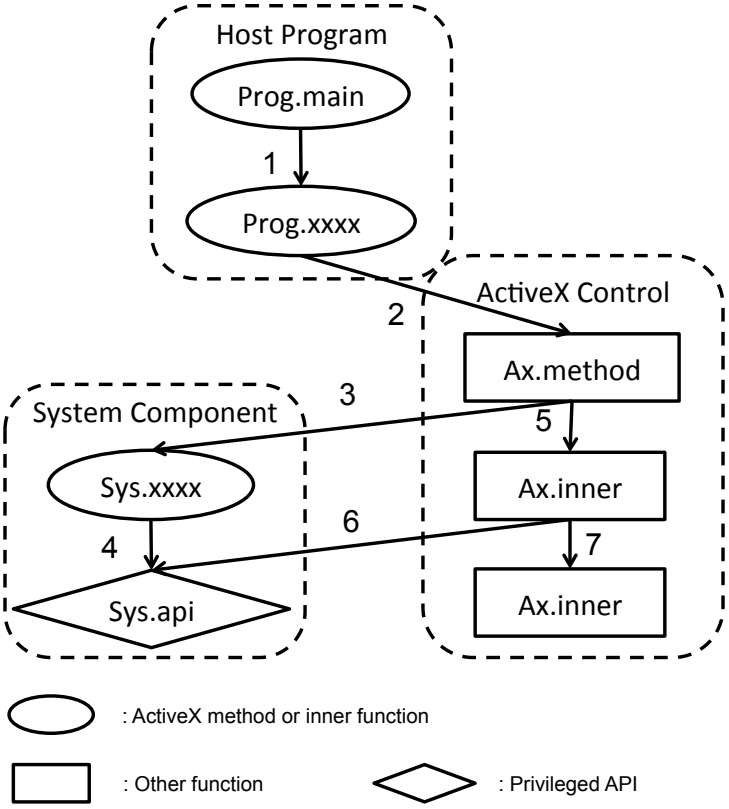
- *Privileged APIs* are system calls in Windows that grant the access to various system resources. For example, `NtWriteFile` is a privileged API to write files in Windows.
- *Other functions* are neither entry functions nor privileged APIs in ActiveX PRM. They are defined for convenience. They can be any functions in the components other than ActiveX.

Examples of the various kinds of functions are shown in Figure 4.5a. The dashed boxes in the figure group these functions into different components.

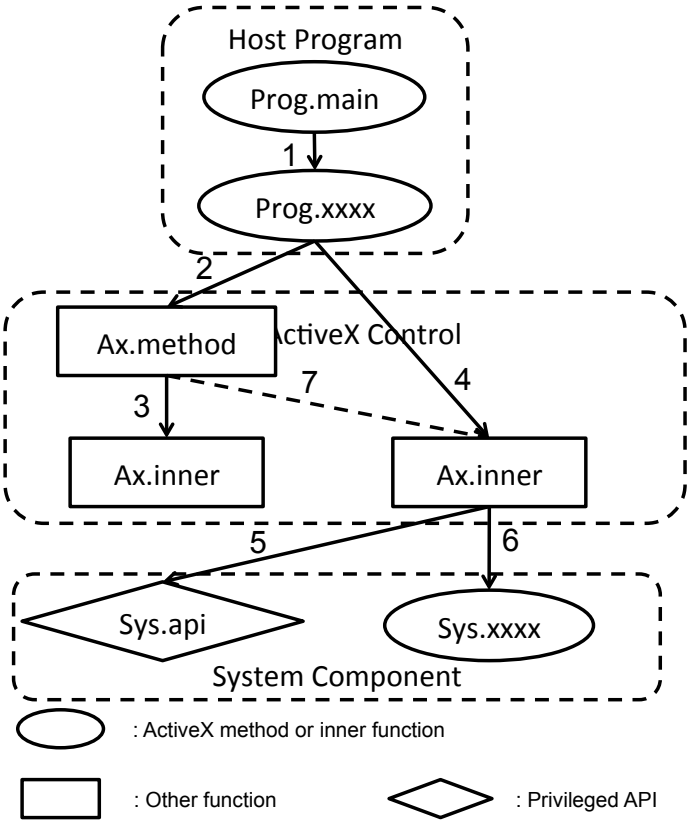
There are two types of access path in the ActiveX PRM as shown in Figure 4.5:

- A *direct access path* of ActiveX method m is a path from the root node r (the entry function of the program, e.g. main) through an ActiveX method m to a privileged API. There can be ActiveX inner and other functions along the path, such that m is the first ActiveX methods along the path, and there are no ActiveX inner functions from r to m. There are two direct access paths in Figure 4.5a: one of them is through edges 1, 2, 3, 4 and the other is through edges 1, 2, 5, 6.
- An *indirect access path*, on the other hand, is a path from the root node r through an ActiveX inner function f to the privileged APIs, such that f is the first ActiveX inner function in the path and there are no ActiveX methods from r to f. In Figure 4.5b, the path comprises of the edges 1, 4, 5, is an indirect access path.

Intuitively, a direct access path of ActiveX method m describes the API misuse of m in ActiveX directly. However, there are subtle interactions between the ActiveX controls and the hosting program, such that there is no direct path that can be identified in the PRM. This subtle interaction is a use of callback functions. In some ActiveX controls, an ActiveX method m can first register an ActiveX inner function f as a callback function. When certain conditions are met in the host program, e.g. a download is finished, the callback function (which is the ActiveX inner function) is invoked by the host program directly to access the privilege APIs. More details on using ActiveX callbacks are discussed with the analysis of Snapshot Viewer



(a) Direct Access Path



(b) Indirect Access Path

Figure 4.5: Two Types of Access Paths in ActiveX PRM

Table 4.1: Privilege Mapping in ActiveX Models

Privilege set type	API
FILE_ACCESS	NtCreateFile NtWriteFile
PROCESS_CREATE	NtCreateProcess NtCreateProcessEx
LIB_LOAD	LoadLibrary LoadLibraryEx

ActiveX control in Section 4.5.1. Thus, an indirect access path captures the use of callbacks, as the API misuse is caused by m through using f.

In summary, an API misuse vulnerability of an ActiveX method m is described in the PRM by a direct access path from m, or an indirect access path from an ActiveX inner function f to the privileged APIs.

4.3.3 API Misuse Vulnerability Identification

Given a service component and a client component, an API misuse vulnerability in the service component can be detected by determining whether the client component's privilege has been increased through the service component. We do this by searching for access paths in the PRM that lead to privileges not given to the client component. For ActiveX controls, we focus on the privileged APIs shown in Table 4.1. We have categorized the privileged APIs into three groups:[4] FILE_ACCESS (file access), PROCESS_CREATE (process creation) and LIB_LOAD (library loading). They present the privileges that are restricted from the JavaScript in the IE sandbox.

Challenges in ActiveX controls. Although searching for access paths appears to be straightforward, there are some non-trivial complications. For ActiveX controls, the challenge is to identify the ActiveX method, which registers the callback functions in an indirect access path. There are two reasons why this is challenging. First, by the definition of indirect access path, every ActiveX method invoked before the ActiveX inner function from the same ActiveX control in the trace can be the one registering the callback. Second, in the multi-threaded Windows

[4]More can be easily added and these are the ones used in our experiments

environment, there is the possibility of kernel callbacks,[5] which also adds to the complexity of this problem.

We employ a heuristic to identify the ActiveX method registering the callback. First, we identify every ActiveX method invoked before the ActiveX inner function from the same ActiveX in the trace. Then, we add a *"pseudo edge"* from each of these ActiveX methods to the inner function. These pseudo edges form new paths from the ActiveX method to the inner functions and the privileged APIs. We call these paths, *"pseudo path"*. For example, in Figure 4.5b, such a path comprises of edges 1, 2, 7, 5, and edge 7 is a pseudo edge denoted by the dash-lined edge. Finally, we identify these pseudo paths just as how we search for the direct access paths. The ActiveX methods linking the inner functions with pseudo edges are the ones registering the callback functions in these pseudo paths. Manual analysis can be used to reduce the number of pseudo paths. Although our heuristic only gives an approximation, we found that it works well in practice and detects the vulnerability in the Snapshot Viewer ActiveX control, discussed in Section 4.5.1. Other ActiveX controls use direct calls to access system APIs.

4.4 Implementation

We have prototyped our approach on Microsoft Windows XP SP2. We implemented a detection tool to identify API misuse vulnerabilities in ActiveX controls, and a mitigation tool to prevent the privilege escalation attacks from happening in those ActiveX controls. The detection tool is based on a PIN tool [125] to collect the function call/return control flow together with an analyzer to construct the privilege reachability model and identify the vulnerabilities. The mitigation tool contains a proxy to provide access controls of the vulnerable ActiveX methods. In the following paragraphs, we discuss each of them with respect to the design of our approach.

4.4.1 Building a PRM from Traces

We implement an analyzer to build the privilege reachability model from the instruction traces collected by the PIN tool. The trace contains only control flow information for each thread.

[5]Non-local control flow transfers where the Windows kernel calls code in the program, analogous to signals in Unix but this is not due to exceptions.

First, the analyzer needs to recover the function calls by matching the call and return instructions in the trace as mentioned above. More attention needs to be paid for non-local control flows, which can cause the call and return not matching in the traces. Second, the functions in the model are represented by their address offsets in the components, except for the standard exported functions, which have names for the analyzer to determine which components these function calls belong to. The analyzer marks all the ActiveX methods among the functions in the graph, and maps names to them. Note that ActiveX methods are not standard exported functions; rather, they are virtual functions residing in the ActiveX objects and are initialized dynamically on object creation (see Section 4.4.3). The name mapping of ActiveX methods requires using Windows type libraries [11] (analogous to a symbol table). Finally, a privilege reachability model is generated by recovering the call sequences in the traces. In the PRM, every ActiveX method is marked with a name and every ActiveX inner function is presented with an offset in the ActiveX binary.

4.4.2 Finding Access Paths

In order to detect the access paths, the analyzer first marks the privileged APIs in the PRM. Then, it directly searches for access paths. When encountering an indirect access path, the analyzer attempts to add a pseudo edge and searches for the ActiveX method registering the callback functions. A heuristic is used to identify the ActiveX method, which registers callback functions. In this strategy, only one ActiveX method is invoked in the test case. Based on the causality between this single ActiveX method and the inner function in the indirect access path found, it is reasonable to conclude that this ActiveX method registers the inner function. This strategy can be extended to allow more than one ActiveX method invocations in the test case, where the assumption becomes more relaxed.

Library Loading Analysis. An ActiveX method from one ActiveX control can expose another ActiveX component to the program, giving the program the complete access to the exposed ActiveX control. This is similar to allowing a program to load an arbitrary library, which is also a typical threat from an API misuse vulnerability in ActiveX controls. Instead of directly

accessing the library loading APIs, the ActiveX methods use the COM library to call them indirectly. This means there is no access path can be directly identified between the privileged APIs (library loading APIs) and the ActiveX method. To identify this type of API misuse vulnerabilities, we first analyze the libraries loaded by the COM library. Then, we attempt to identify the relationship between the loaded libraries and the arguments of the ActiveX methods. We extract all the metadata of ActiveX controls in the system from the registry. We compare these data with the arguments used in the library loading APIs. If we can identify a match, we mark the ActiveX method to be vulnerable to load libraries. This analysis can have false negatives. A better way is using data dependency analysis on the loading behaviors, just like what we did on binary loading problems in Chapter 3. Even using this simple technique, we are able to identify `MS06-014` IE `CreateObject` vulnerability [12]. Details are discussed in Section 4.5.

4.4.3 Mitigation of API Misuse Vulnerability in ActiveX Controls

There are two directions to mitigate the damage from API misuse vulnerabilities. Isolation based solutions [57, 102, 126] focus on isolating the host program and the shared components, or enforce clear isolation among shared components themselves. Access control based solutions [97, 118] focus on enforcing certain policies when accessing privileged resources.

Our prevention mechanism is an access control based solution. Figure 4.6 shows an example of the system on the Windows web browser, Internet Explorer (IE). Its core component is a proxy, which intercepts the ActiveX controls and the COM libraries. Hooks are applied to monitor the creation and the invocation of the ActiveX controls. The proxy blacklists dangerous ActiveX methods, which are reported by our detection mechanism. Access policies are predefined by studying the API misuse behavior in the PRM model. The proxy enforces the policies and determines whether to allow or deny the access to each ActiveX method from the host program, e.g. Internet Explorer. We also implement a loader based on the standard API `CreateRemoteThread` to load the proxy into IE. The loader provides flexible deployment of the proxy library.

The proxy denies any invocation of vulnerable ActiveX methods in the blacklist. Rejection of the invocation is through raising an `E_ACCESSDENIED` exception, i.e. *General Access*

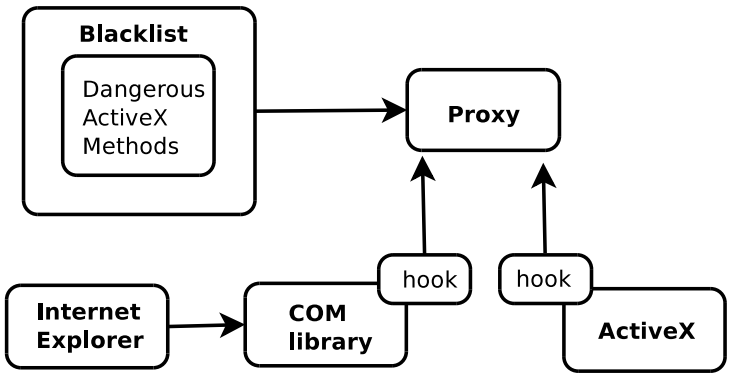


Figure 4.6: Mitigation of API-misuse in the Web Browser

Denied exception used in Windows to block access to certain functionality. The advantage of this exception is that it does not affect the use of other methods in the hosting program, such as web browsers. The ActiveX controls can still function as long as the vulnerable methods are not invoked. In this way, the usability of the ActiveX component is preserved in a fine-grained manner, overcoming the coarse-grained security mechanisms discussed in Section 2.1.1.

ActiveX Hooking. Intuitively, intercepting exported methods is through direct hooking the exported function table in the shared libraries in Windows. However, this is not applicable for ActiveX controls, because ActiveX methods are virtual functions in ActiveX objects that are dynamically invoked. We need to dynamically hook the ActiveX methods when the ActiveX objects are created.

To invoke a method in an ActiveX control, three steps are needed: (i) creating an ActiveX object using the COM library API `CoGetClassObject` or `CoCreateInstance`; (ii) getting the object from the `IClassFactory` interface using the virtual function `CreateInstance`; (iii) invoking the method by using the virtual function `GetIdsOfNames` and `Invoke`.

Accordingly, we perform ActiveX hooking following these steps:

- We intercept the standard exported functions `CoGetClassObject` and `CoCreateInstance` in the COM library. We retrieve the ActiveX object returned by these APIs.
- We extract the inner objects from the ActiveX object and the virtual function table. We hook the virtual function `CreateInstance`, and retrieve the `IClassFactory` interface.

Table 4.2: Number of methods with critical access paths in six ActiveX controls

ActiveX Controls	Total methods	file operation	library loading	process creation	access paths
MS ADODB Stream	26	2	0	0	2
MS RDS DataSpace	3	0	1	0	1
MS Office Snapshot Viewer	27	2	3	0	3
Chilkat Crypt	159	2	1	0	2
InstallShield Update Service	14	6	3	3	8
Zenturi ProgramChecker	23	9	4	0	9

- We extract `GetIdsOfNames` and `Invoke` by identifying their fixed offset in the virtual function table from the `IClassFactory` interface. We intercept these functions to monitor the ActiveX methods invocations from the scriptable interfaces.

The hooks are implemented in a way similar to Microsoft Detours implementation [13]. By performing ActiveX hooking, we can intercept the method invocations directly and block the access to privileged APIs.

4.5 Evaluation

We evaluated our approach using real-world ActiveX controls. We perform the experiments in Windows XP SP2 with Internet Explorer as the hosting program. The ActiveX controls are *MS ADODB Stream* (ADODB), *MS RDS DataSpace* (RDS), *MS Office Snapshot Viewer* (Snapshot), *Chilkat Crypt* (Chilkat), *InstallShield Update Service* (InstallShield) and *Zenturi ProgramChecker* (Zenturi). All controls have proper documented specifications except InstallShield and Zenturi. We introduce these controls as follows:

1. *MS ADODB Stream ActiveX Control.* This control (*ADODB.Stream*) is provided as part of the ActiveX Data Objects (ADO), which is installed by default in Windows. It is designed to manipulate binary streams of data in a file system or an e-mail system. One method, `SaveToFile`, presents the ability to save binary data to local file system silently. SecurityFocus [14] describes this vulnerability as a file installation weakness.

2. *MS RDS DataSpace ActiveX Control.* This control (*RDS.DataSpace*) is provided as part of the ActiveX Data Objects (ADO) and is distributed in Microsoft Data Access Com-

ponents, which is installed in Windows by default. This ActiveX is designed to create client-side proxies to transfer application data across process or machine boundaries. One of its methods, `CreateObject`, presents the ability to load arbitrary libraries and bypass the IE ActiveX loading checks. `MS06-014` [12] and `CVE2006-0003` [15] describe this vulnerability as remote code execution.

3. *MS Office Snapshot Viewer ActiveX Control.* This ActiveX control (*snpvw.Snapshot Viewer Control.1*) is a component of MS Office Snapshot Viewer which is installed with MS Office Access 2003. This ActiveX is designed to view a snapshot report, which is a print preview, from IE and other applications. There are two methods, `SnapshotPath` and `CompressedPath` that allow an attacker to download file to local file system silently. `MS08-041` [16] and `CVE2008-2463` [17] describe this vulnerability as arbitrary file download.

4. *Chilkat Crypt ActiveX Component.* This ActiveX control (*ChilkatCrypt2.ChilkatCrypt-2.1*) is a software component for encrypting and decrypting strings and binary data with various encryption algorithms. One of the functionality through `WriteFile` silently writes file to local file system without user intention. `CVE2008-5002` [18] describes this vulnerability as arbitrary file creation and overwrites.

5. *InstallShield Update Service ActiveX Control.* This ActiveX control (*DWUSWebAgent-.WebAgent.1*) is an optional component for MacroVision InstallShield, which is for software vendors to deliver and setup their product on the end-user systems. The control grants the vendors the ability to notify and install product patches and updates. There is a method, `DownloadAndExecute`, which allows attackers to download and execute arbitrary code. `CVE2007-5660` [19] describes this vulnerability as remote code execution.

6. *Zenturi ProgramChecker ActiveX Control.* This ActiveX control (*SafeAndSoundATL-.NixonConfigMgrEx.1*) grants the end users the ability to control the applications running on the local system. It grants the applications the trust from the users and delivers real-time protection to prevent unauthorized programs from executing. These are two meth-

ods `DownloadFile` and `NavigateUrl` that allow both arbitrary file download and remote code execution described in `CVE2007-2987` [20] and `EDB-ID 4050` from exploit-db [21].

4.5.1 Effectiveness Evaluation

Our evaluation uses test cases from the ActiveX documentation, and test templates described in Section 4.3.1 if no documentation is available. (For the third party ActiveX controls evaluated, no documentation is available.) Privileged APIs belong to the three privilege sets in Table 4.1. ActiveX inner functions are presented as `library.offset`. For example, `snapview.c5ad` denotes the function at offset `c5ad` in the Snapshot Viewer ActiveX control.

Table 4.2 shows the overall results of the API misuse vulnerabilities detected. For each control, we list the total number of the exposed methods and number of the exposed methods that have access paths found. The access paths are further broken down according to whether they involve file access, library loading or process creation APIs. Some methods are counted in multiple privilege sets, so the total number of access paths in the three categories may exceed sum of the number of methods with access paths. Some ActiveX can provide much more methods than others, e.g. Chilkat Crypt provides 159 methods, and RDS DataSpace only has 3 methods. Five ActiveX controls have access paths to file system calls. And these paths are the majority of all the access paths. This indicates that file system access is usually included when designing an ActiveX control. Library loading privileges are also commonly provided in these ActiveX controls. Only the InstallShield ActiveX control can access the process creation API to execute the update downloaded. In summary, we identified 25 access paths in total, and seven indirect access paths. All documented vulnerabilities in the six controls are detected by our approach.

We now describe three representative ActiveX controls among them.

Snapshot Viewer ActiveX Control. This ActiveX control has 27 methods that are available to the hosting program. The generated model has 4963 nodes. We found three methods have access paths to the privileged APIs. The `SnapshotPath` and `CompressedPath` methods specify the path to the snapshot file to be displayed in the Snapshot Viewer. We

found both of them have access paths to `NtCreateFile` and `NtWriteFile`. With a local URL as the argument, `CompressedPath` has a direct access path to `NtCreateFile` and `NtWriteFile`. With a remote URL as the argument, `SnapshotPath` has indirect access path to the same APIs. `PrintSnapshot` together with the above two methods have a direct access path to `LoadLibraryExW`.

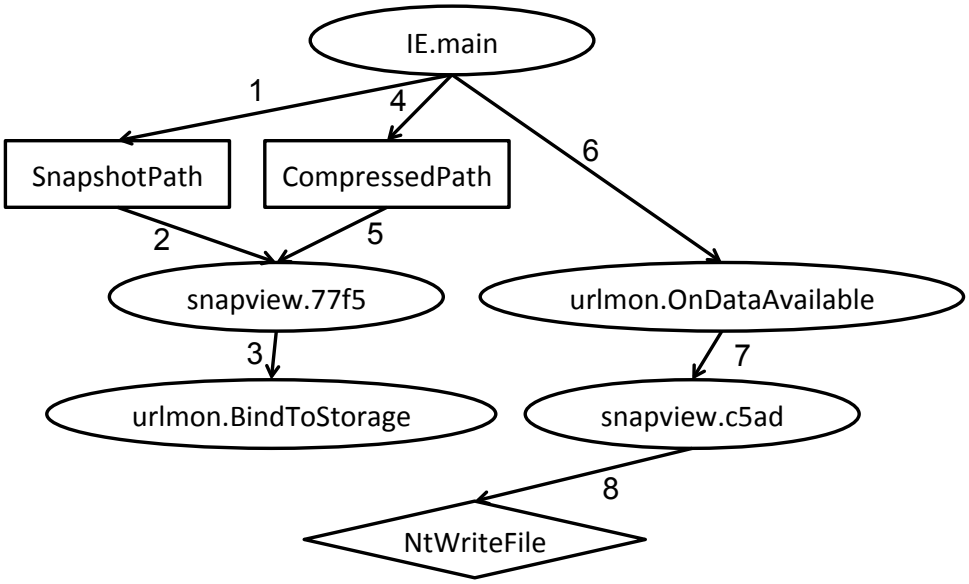


Figure 4.7: Indirect Access Path Using Callback Function `snapview.c5ad`

When searching the access paths in the model, we identify the ActiveX inner function `snapview.c5ad` as the callback function in an indirect access path (edges 6, 7, 8) shown in Figure 4.7. The PRM in Figure 4.7 only shows important functions. We illustrate how the attack in the motivating example works as follows:

- `SnapshotPath` calls `snapview.77f5`, which registers the callback function `snapview.c5ad` through `urlmon.BindToStorage`. (edges 1, 2, 3)
- `CompressedPath` calls `snapview.77f5` to record the path to write the payload. (edges 4, 5)
- When the payload is downloaded, `urlmon.OnDataAvailable` signals the download finished event resulting in the callback function `snapview.c5ad` invoked to write the payload to the local file system. (edges 6, 7, 8)

Together with the vulnerable ActiveX methods, we identified the callback function `snapview.c5ad` and the whole control flow, to understand how the misuse happens.

ADODB Stream ActiveX control. We use test cases in the documentation to manipulate the *Stream* object in this ActiveX control. We identify 2 out of 26 methods having access paths directly to `NtCreateFile`. The `SaveToFile` method is to save the binary content of a *Stream* object to a file. The `LoadFromFile` method is to load the contents of an existing file into a *Stream* object. Further inspection on the access path and `SaveToFile` shows that this method first invokes `NtCreateFile` to create a file with the `FileName` argument. Then, it invokes `NtWriteFile` to write the data inside *Stream* object into the file. The `SaveOptions` argument can be set to arbitrarily overwrite an existing file silently.

RDS DataSpace ActiveX Control. We found that one of the three methods in this ActiveX control has an access path to the privileged APIs. The `CreateObject` method is to create various business objects either in a remote server or in a local process. It can load any library and create the object registered in the local system. We found it has access paths to `LoadLibraryExW` for loading libraries, including other ActiveX controls. This vulnerability is described in `MS06-014` [12], which allows arbitrary library loading and bypasses the killbit checking of Internet Explorer on ActiveX controls.

Figure 4.8: Call Sequences to Load `adodb.stream` and `wscript.shell`

In a sample attack exploiting this vulnerability, `CreateObject` is used to create objects from disabled vulnerable ActiveX controls in IE. The newly created object can be further leveraged to achieve remote code execution. We observe that `ole32.CoCreateInstance` loads libraries through invoking `LoadLibraryExW` and then creates the objects. `CreateObject`

creates `ADODB.Stream` and `WScript.Shell` by loading binaries, `msado15.dll` and `wshom.ocx`. These objects are, by default, disabled ActiveX controls in IE, but the use of `CreateObject` bypasses the restriction. `ADODB.Stream` is used to write the arbitrary file to the local storage, and `WScript.Shell` can execute the payload. Figure 4.8 shows the important functions in the call sequences where library loading API loads the two objects. Both the sequences starts from the main function of IE, via the `CreateObject` method, library loading API `LoadLibraryExW`, to the entry points of the binaries.

Effectiveness of the Mitigation. We evaluate the effectiveness of the proxy in the prevention mechanism. The blacklist consists of 11 methods from the six ActiveX controls described above. All of them have API misuse vulnerabilities. By running the test cases we find that all methods in the blacklist are successfully blocked, while other methods in the ActiveX controls are still functioning. We preserve the normal functionality while blocking only the potentially vulnerable ActiveX methods in the prevention mechanism.

4.5.2 Performance Evaluation

To evaluate the performance of the detection approach, we selected 27 test cases from Office Snapshot Viewer ActiveX control. Each method is tested separately in a new IE process. The total time for instrumentation is 1174 seconds (43.5 seconds/test case). The total time for building the model and searching of access paths is 264 seconds (9.8 seconds/test case). Our implementation is only a prototype and is not optimized. The time to find vulnerabilities is still reasonable.

To evaluate the overhead of the prevention mechanism, we selected 12 test cases from three ActiveX controls, and tested them separately in a new IE process. We tested ActiveX methods that are both in and not in the blacklist. The overheads range from 0.01-1.7%. This show that the access control based proxy is efficient with negligible overhead.

4.6 Related Work

Existing ActiveX security mechanisms in Windows is mainly based on trust. Internet Explorer trusts ActiveX controls installed locally in the Windows system except those blocked by compatibility flags in the registry (*killbits*). For remote ActiveX controls, the user provides a white list of trusted sites and permit remote ActiveX controls from the white list. For untrusted ActiveX controls, IE asks the user for permission to use the control. These security mechanisms rely on the trust from the users, however, the users usually do not know whether the ActiveX controls are safe or not. Once vulnerability in an ActiveX control is known, the typical solution is to completely block it but that means that all the functionality performed by the control in IE is lost.

Research works in ActiveX security mainly focus on memory vulnerabilities. Dromann and Plakosh [52] propose an automated fuzzing system to detect security flaws in ActiveX controls. Its target is memory-related vulnerabilities, instead of API misuse vulnerabilities. Song et al. [110] propose an approach to detect malicious exploitation of vulnerable ActiveX controls to prevent drive-by download attacks. The prototype prevention is integrated into IE with ActiveX hooking, using similar techniques as our proxy to block dangerous methods in ActiveX controls.

The reachability analysis in our approach is related to approaches using model checking and system graph analysis. Schneider [107] proposes security automata for defining security properties and preventing the illegal actions in the system. MOPS [40] detects attacks by checking the reachability of a state that violates the desired security goal in a model. Both Sheyner et al. [108] and Jha et al. [71] construct attack graphs for model checking to detect safety violation in the system. The attack graphs make use of network traffic. Backtracker [75] identifies the files or processes that cause an attack through dependencies between these files and processes in a system-level dependency graph. As another example, Martignoni et al. [88] perform data-flow analysis to identify high-level actions from system calls.

To address the privilege escalation problem in Android, many solutions are proposed. ComDroid [42] uncovers possible unintended consequences of exposing certain components of Android applications (apps). It only detects vulnerabilities within the component. CHEX [85] ap-

plies the similar idea, but also detects cross-component vulnerabilities in addition. SEFA [123] applies provenance analysis to determine the impact of vendor customizations on the Android component security. It can find in-component, cross-component, and cross-app vulnerabilities. Woodpecker [65] is a prototype system to detect capability leaks in Android-based smartphones. The prototype performs static analysis on the specific control-flow graph for each capability, which includes possible path identification and infeasible path pruning. All these solutions are based on static analysis, which may not provide complete results when applying on systems with a large amount of dynamical invocations.

Mitigation of privilege escalation among software components relies on enforcement of access control policies or application of strong isolation constructed among the shared components. Codejail [126] provides isolation between untrusted library components and the main program at the memory level. Addroid [102] proposes an advertising library separation framework to isolate the untrusted advertising library using newly introduced APIs. IPC inspection [57] isolates the Android components using processes in the OS. It reduces the program's permission after receiving the request from a low privileged program. Saint [97] examines the exported interfaces and extends the Android framework to enforce inter-application security policy both at install time and runtime. Compac [118] provides a fine-grained access control at component level in Android. It extends the Android framework to enforce and check these component level permissions.

4.7 Summary

In this chapter, we present a system to detect and prevent API misuse vulnerabilities in shared components, focusing on the problems with ActiveX controls in Windows. We detect vulnerable ActiveX methods in ActiveX controls by using a reachability analysis. We believe that our system can be adapted to other systems using shared components, such as Android. We provide an access control based mitigation solution to block vulnerable ActiveX methods in the hosting programs. We are able to identify API misuse vulnerabilities in six real-world ActiveX controls. The cost of the detection is reasonable and the overhead of the mitigation is negligi-

ble. By applying our solution in this chapter, we explain the interactions among the software components with a model to avoid the misunderstanding that causes component misuses. This solution also provides an effective mitigation of component misuses and preserves the usability of the normal functionalities.

We plan to track various forms of data flows as we did in detecting binary loading vulnerabilities. With the data flow tracking, we attempt to detect more forms of ActiveX API misuses, e.g. the library loading misuse discussed in Section 4.4.2. With just a simple form of data dependency analysis, we can identify how the IE ActiveX blocking mechanism is bypassed by leveraging the library loading APIs. We also plan to support automatic test case generation in ActiveX API misuse tracking mechanism. Currently, the test cases are generated for JavaScript in the IE browser. We plan to support more scripts, such as VBScript, and more Windows programs, such as Office programs, as hosting programs for ActiveX controls.

Chapter 5

Preventing the Misuse of Components in Android ICC

5.1 Introduction

After having a better understanding on the causes of the component misuse problem in the previous chapters, we propose a solution to secure the interactions between software components. We target the Android platform where the misuse of components are common and cannot be prevent using a blocking mechanism as we did in preventing API misuse. Instead, we propose a different mechanism to both prevent component misuse and preserve the usability of the software components.

Android protects applications (apps) using a permission-based system. Each app is assigned a set of permissions at the installation time. They are isolated from each other and can only access the resources guarded by the assigned permissions at run time. To support app interactions, Android provides several inter-component communication (ICC) mechanisms, providing extended functionalities to an app through communicating with another app. The *Intent* is the main mechanism used in Android ICC, which provides specific structured messages across the Android system.

An app (the sender) can send an *intent* to another app (the recipient) to request access to services offered by the recipient. The sender does not need to possess the permissions required

by the recipient to complete the service. Rather, the recipient exercises its own permissions *on behalf of* the sender. This results in a form of privilege escalation [49], called *permission re-delegation*, which is a component misuse problem in the Android component communication model, caused by complicated interactions among software components. In Android, prevention of permission re-delegation relies on the effort of app developers, e.g. they may ask the sender to provide proper permissions in the ICC, but no system-wide mechanisms are given. Existing solutions [51, 57, 93] are proposed to prevent permission re-delegation by enforcing tightened permissions. However, the tightened permissions can break the functionality of the recipient, which unnecessarily renders these prevention mechanisms less usable.

We propose a framework, called UPPDROID, to prevent permission re-delegation while at the same time maintaining the usability of Android ICCs. Our approach differs from existing solutions: instead of enforcing tightened permissions, we allow access of a virtual resource. We enhance Android ICCs with provenance which is used to determine the resource access rights of the recipient. Resources for which privilege escalation can occur in the sender are virtualized so that the recipient uses those virtualized resources. In order to do this, our virtualization needs to be aware of the specific ICC context and also maintain consistency of the resource data with respect to operations performed on the resource. As virtualized resources can be accessed by the recipient, the advantage of UPPDROID is that it provides better usability in that the apps continue to work albeit with virtual resources. This is an improvement over existing solutions that tend to break apps, e.g., causing them to crash due to lack of permissions. Moreover, the security of UPPDROID can be configured with a trade-off between the security and the usability.

The contributions of UPPDROID are in the following aspects:

- We prevent the component misuse caused by complicated component interactions in a specific communication model in Android, namely, permission re-delegation in Android ICC.
- We focus on preventing usability problem due to the security mechanism causing problem to the apps, e.g., crashing.
- We provide a mechanism to enhance the Android ICC to carry ICC contexts and various information between apps.

In the following sections, we discuss the basic Android ICC with permission re-delegation in Section 5.2. We describe the design and implementation of UPPDROID in Section 5.3. We show the performance overhead and the case studies in Section 5.4. Finally, we discuss the related work and conclude the chapter in Section 5.5 and 5.6.

5.2 Overview

5.2.1 Android Component Interaction

Android Components and ICC. Recall from Chapter 2, Android apps are comprised of four types of components, designed with specific purposes. The *Activity* component interacts with other components through a GUI, e.g., touchscreen. In an app, each displayed screen is usually an activity. Only one activity is active at a time. The *Service* component provides background processing and supports remote procedure calls and callbacks. The *Broadcast Receiver* component receives system-wide broadcast announcements that are sent by both the system and apps. It supports asynchronous event notifications to other apps. The *Content Provider* component provides access to database of Android apps with SQL-like APIs.

Android ICC provides apps with extended functionality to allow apps to make use of other apps that provide extra functionality. An intent is a message that serves as an abstract description of actions for the recipient to carry out during an ICC. More specifically, intents are used to start and interact with Android components using corresponding APIs, e.g., *startActivity(Intent)* which is used to start and interact with an activity. Intents can be utilized either in an explicit or implicit way. Explicitly, an intent can specify the recipient by name. Implicitly, the Android platform decides which apps can receive the intent.

Resources and Permissions. Android protects its resources with a permission-based mechanism. Each app explicitly claims its permissions to access various system resources in a manifest file at the install time. When resources are accessed by the app, Android checks whether the app has the corresponding permissions. Within an app, all components share the same permissions. There are over 150 permissions guarding various Android resources since Android 4. These resources are from the framework level and kernel level. At the framework level,

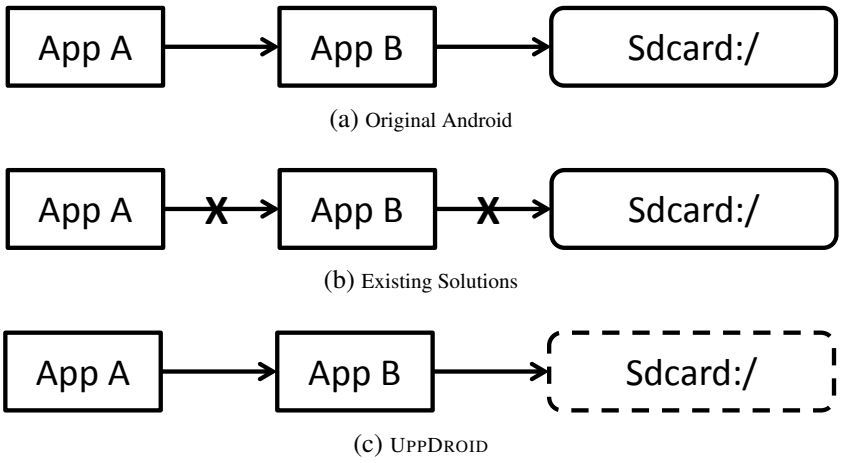
Android provides device-specific system services, such as telephony, location and camera, as well as various content providers, such as contacts, call logs, and SMS. At the kernel level, system resources managed with the Linux security mechanism include the file system and network sockets. For example, *sdcard* is a resource managed at this level.

Permission Re-delegation in Android ICC. In Android ICC, the recipient can exercise its own permissions on behalf of the sender, which makes the sender gain additional privileges. This is called *permission re-delegation*, where the sender indirectly obtains the permissions from the recipient. This can be caused by intentional functionality of the recipient app, by accidental exposure of internal functionality to another app, or even by incorrect implementation by the developers. Permission re-delegation is very common in Android ICC.

For example, an image-processing app A (without the permission *WRITE_EXTERNAL_STORAGE*) requests a file manager app B (with the *WRITE_EXTERNAL_STORAGE* permission) to process a photo on the *sdcard*, through public APIs, such as *ReadFile* and *WriteFile*. This is a typical manifestation of permission re-delegation, where A gains privilege of accessing *sdcard* from B. Note that users can be involved in this ICC, e.g., opening the file manager from the image-processing app by clicking a certain button. Consider a malicious app also without the *WRITE_EXTERNAL_STORAGE* permission. It wants to gather private data and execute malicious payloads. By using the file manager app, private data on the *sdcard* can be gathered, and malicious payloads can be generated and saved to the *sdcard*. Thus, the permission re-delegation in this example leads to sensitive data leakage and malicious code execution. Many such problems have been identified. For example, Felt et al. [57] found 15 permission re-delegation problems in 5 system apps in 2011. Grace et al. [65] found 8 phone images including 11 permissions have similar privilege escalation problem. Not only the malicious code can be executed in this way, but also the sensitive private information can be leaked through permission re-delegation.

5.2.2 Approach Overview

We show the overview of UPPDROID and discuss the privilege escalation problem in permission re-delegation and the drawbacks of existing solutions with examples.



(a) Original Android

(b) Existing Solutions

(c) UPPDROID

Figure 5.1: Examples

In Figure 5.1, the same image-processing app A (without the permission *WRITE_EXTERNAL_STORAGE*) requests a file manager app B (with the *WRITE_EXTERNAL_STORAGE* permission) to process a photo on the *sdcard*. In the original Android system (Figure 5.1a), this ICC is allowed by design, so B accesses the *sdcard* on behalf of A. However, this permission re-delegation allows A to indirectly access the *sdcard*, as if it has the *WRITE_EXTERNAL_STORAGE* permission. If A is acting maliciously, it can erase *sdcard* through the permission re-delegation, if B provides file deletion.

Existing solutions [51, 57, 93] based on the principle of least privilege are proposed to prevent privilege escalation caused by permission re-delegation. They can completely block the ICC by not starting B (left side arrow in Figure 5.1b). Alternatively, they run B with the removal of permissions (right side arrow in Figure 5.1b), which results in B being run with the intersection of permissions from both A and B, e.g., the *WRITE_EXTERNAL_STORAGE* permission is removed from B. With the removal of the permission, A cannot gain additional privileges from this ICC. However, forcing B to run without the *WRITE_EXTERNAL_STORAGE* permission may crash B. In practice, this is quite likely to occur as an exception will occur. Thus, the functionality of B is lost. In this chapter, we call such solutions, "denial-based" solutions. They are less reliable [74], especially, when the recipient is also servicing other apps, as they break the assumption that the permissions in the Android app manifest can always be granted. The developers may not have considered handling the exceptions caused by permission removal in the first place. The consequences of the crash in these solutions affect

the system in many ways, especially, (i). the crash is usually the whole app, including all its activities. These activities can be interacting with other apps. The data or the program states can be damaged due to the app crash; (ii). recovering from the crash can be demanding. The usual way is to restart the whole app from crashes. This can not only consume many system resources and power on a mobile device, but also require many user interactions to reach the state before the crash happens.

Our goal in this chapter is to prevent privilege escalation in Android ICC while still making the app usable by avoiding the app crashes due to the permission removal. We propose UPPDROID, which employs dynamic resource virtualization on the system resources guarded by permissions that are limited or removed in the "denial-based" solutions. It allows the Android ICC by creating virtualized resources for the recipient to access based on the permissions of the apps involved in the Android ICC. The system resources are virtualized in a way that no privileges are leaked to the sender. Since no permission is removed from the recipient, the functionality is preserved and no crash is caused by removing the permissions. As shown in Figure 5.1c, UPPDROID determines that the system resource *sdcard* is to be virtualized because the intersection of permissions from A and B does not contain *WRITE_EXTERNAL_STORAGE*. UPPDROID creates a virtualized *sdcard* for B to access on behalf of A. A does not gain any privilege from this ICC since the *sdcard* is not the real *sdcard*.

UPPDROID aims to prevent the privilege escalation caused by permission re-delegation in Android ICC. If the sender has more permissions or at least the same permissions as the recipient, the ICC is allowed since no privilege escalation is caused. UPPDROID also supports security level customization. If the user intends to use certain Android ICC, UPPDROID can be set to allow this ICC as well as informing the user of potential privilege escalation risks. UPPDROID focuses on protecting the resources guarded by the default Android permissions. Although customized permissions in third party apps are not the main focus, UPPDROID can be extended to cover them.

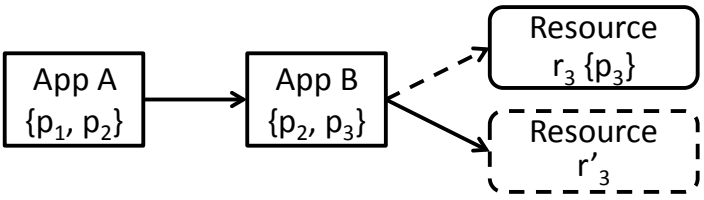


Figure 5.2: Permission re-delegation prevention in UPPDROID

5.3 Design & Implementation

We now describe the design of UPPDROID. The basic idea of how UPPDROID prevents privilege escalation is shown in Figure 5.2. Suppose App A with permission set $\{p_1, p_2\}$ sends an intent to access a resource r_3 which can be met by App B with permission set $\{p_2, p_3\}$. The "denial-based" solutions block this access shown in Figure 5.2 (dashed arrow), as after the intersection of the permissions (p_2) is applied on App B, App B only has p_2 and thus cannot access r_3. In UPPDROID, we still run App B with its original permission set as if p_3 is "granted". However, instead of accessing r_3 directly, we virtualize it as r'_3 as shown in Figure 5.2 (dashed rounded box). App A gains no additional privilege from this Android ICC even though the permission p_3 is granted during our virtualization of r'_3.

We define following rules in UPPDROID for our permission re-delegation prevention mechanism. We let $P_t(A)$ denote the set of permissions held by App A at time t, $R(P_t(A))$ denote the resources guarded by those permissions and $RV_t(A)$ denote the resources that are virtualized at time t. We have the following rules:

- Initial state: $RV_t(A) = \emptyset$, $P_t(A) = P_{original}(A)$. When App A starts, it has the original permissions of A as granted by the user. No resources are virtualized.
- Resource virtualization for recipient: In an ICC from A to B at the time t, then $RV_t(B) = R(P_{t-1}(B) - P_{t-1}(A) \cap P_{t-1}(B))$. We virtualize the resources which are guarded by the current permissions of receiver B but which are not in the intersection of current permissions from the sender A and the receiver B. The resources which are virtualized are a subset of the resources an app originally has access to.
- Sender's and receiver's permissions remain unchanged: In an ICC from A to B, we have $P_t(A) = P_{t-1}(A)$ and $P_t(B) = P_{t-1}(B)$, thus permissions of the apps are unchanged.

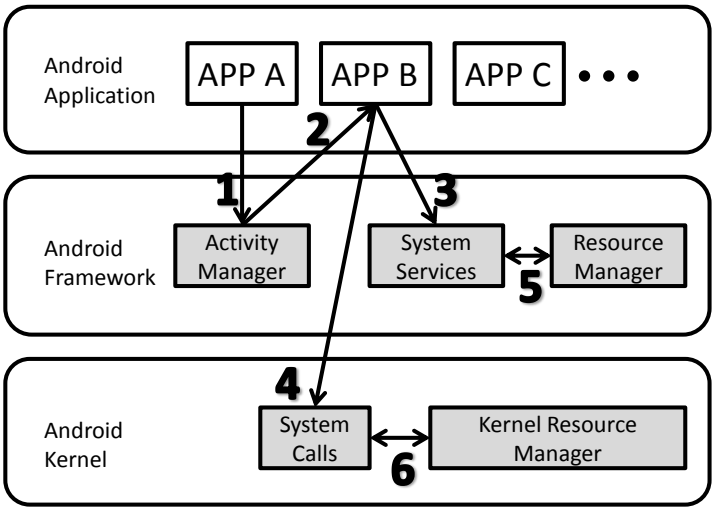


Figure 5.3: Architecture of UPPDROID

However in cases where resources are not easily virtualizable, then $P_t(B) = P_{t-1}(A) \cap P_{t-1}(B)$.

The implication of enforcing the above rules is that in order to maintain the functionality of the recipient app, its original permissions are used as much as possible. These rules are enforced in UPPDROID by default. If users want an Android ICC to happen as the original Android system, where privilege escalation can happen, UPPDROID also can alert the user of potential risks in this ICC and allows the ICC ensuring the intention from the users, e.g. a pop-up for approval. Moreover, the resource virtualization can also be customized to relax the default rules. We discuss these features in Section 5.3.2.

Figure 5.3 shows how UPPDROID works when an ICC happens, e.g. App B delegates the access to resource *r* on behalf of App A. In the diagram, arrows denote the individual step that happens when an ICC is issued. We create a *ResourceManager* and a *Kernel ResourceManager* to manage the resource accesses at the framework level and the kernel level. Steps 1, 2 and 3 describe the ICC to access system resources, whereas steps 1, 2 and 4 describe the access to kernel level resources. In step 1, App A creates an intent *i* which will eventually cause resource *r* to be accessed by app 2 and intent *i* is passed to *ActivityManager*. In step 2, the *ActivityManager* starts App B[1], which provides the access to *r*. The *ActivityManager* also passes the ICC information from App A in the intent *i* to App B, and records this information

[1]In original Android, when App A calls *startActivity* to start App B, App B is not aware of App A. The ICC is transparently handled by the *ActivityManager*.

in our *Kernel ResourceManager*. In step 3, in order to access *r*, App B invokes a system service manager, e.g. *LocationManager*. It passes the ICC information in *i* to the system service manager. To access a resource managed by kernel, step 4 shows where App B makes system calls to the kernel, e.g. to access the file system. In step 5, the system service manager interacts with the *ResourceManager* to decide the virtualization which is dependent on the ICC information. In step 6, when App B accesses resources at the kernel level, the intercepted system calls interacts with our *Kernel ResourceManager* to virtualize the kernel resources.

In summary, UPPDROID achieves the goal with the following features:

- *transparent ICC provenance*: the ICC information is generated and handled at system level to provide ICC provenance. The information is protected in the Android system which is not accessible by the apps.

- *customizable resource virtualization*: the resource virtualization can be customized. Besides the rules discussed previously, which enforce intersections of permissions, the resource virtualization supports choices of users to always allow certain Android ICC or relax the restrictions to certain extent.

We describe them with details in the following discussion.

5.3.1 Adding ICC Provenance

We first describe the notion of an Android ICC chain. In App A, a component can start another component in App B through an intent. We denote this by the notation $A \rightarrow B$. In the same fashion, App B can start another component in App C, represented by $A \rightarrow B \rightarrow C$. We call $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n$ an *ICC chain* where A_i are apps. In fact, they are the components in A_i which are involved (invoked) in the ICC but a component can always be mapped to an app, i.e. $a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_n$ where a_i is a component in A_i. Notice that an ICC chain is always finite; it may have repetition in A_i corresponding to a form of recursion. For example, if a component a_1 in App A calls a_2 in App A, and a_2 calls b_1 in App B which calls c_1 in App C, the ICC chain for the interactions is $A \rightarrow A \rightarrow B \rightarrow C$. The launcher app which is a system app to start other apps through the GUI is usually the first component in the ICC chain. As it is a privileged

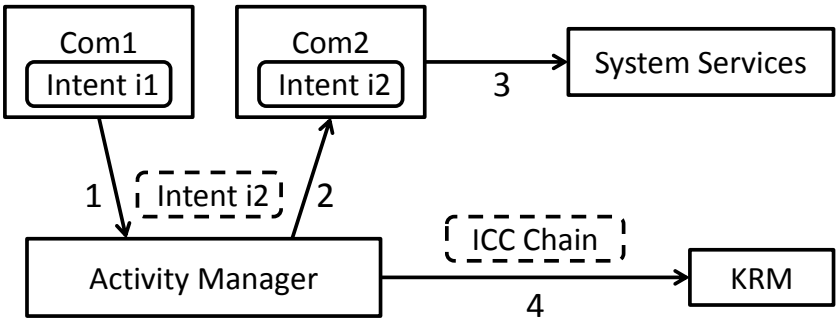


Figure 5.4: ICC Chain Passing Mechanism in UPPDROID

system app, permission re-delegation is not an issue. In UPPDROID, we exclude the launcher app from the constructed ICC chain.

In Android, no ICC provenance information is maintained due to the indirection and message dispatching. The recipient is not aware of the sender's identity through *startActivity* API. Only explicit usage of *startActivityforResult* can provide the recipient with the notion of the caller, however it is not always used, and it relies on the recipient to take actions, e.g. preventing permission re-delegation. We design an ICC provenance mechanism to carry ICC chains as contexts for virtualization. Although there are existing solutions providing ICC provenance in Android like Quire [51] and Scippa [27], our mechanism provides a transparent ICC handling and interacts with the resource virtualization mechanisms to prevent permission re-delegation. It maintains and passes the ICC chain across component boundaries and to the resource managers. We discuss how this is achieved in the following discussion.

Passing Across Component Boundaries. Android components, such as activities, services and broadcast receivers use the Android intent to pass data across component boundaries. Naturally, our ICC mechanism extends the standard Android intent class to pass ICC chains.

In Android, ICCs and component life cycle are managed by the *ActivityManager*. We modify the *ActivityManager* to pass the ICC chains together with the intents. Figure 5.4 shows how ICC chains are passed using intents from component 1 to component 2. In this example, intent i_j denotes the intent that starts the component com_j. In step 1, com_1 sends the intent i_2 to the *ActivityManager*. The *ActivityManager* retrieves intent i_1 from com_1, and extracts the ICC chain in i_1 and stores it in i_2. In step 2, the *ActivityManager* decides to start com_2 which is designated to handle i_2. The *ActivityManager* appends the app information which contains

com_2 to the ICC chain in i_2. In this fashion, The ICC chain information is generated and passed across component boundaries.

There is a special type of intents, called *pending intents*, to let a component request another component to start a third component on behalf of the first component. By sending a pending intent to another component, the sender grants the recipient the right to perform the operation in the intent as if the receiver was the pending intent sender itself (with the same permissions and identity). For example, a component a allows component b to start component c on behalf of a using a pending intent. Component b is using a's permission to start c. For cases like this, we treat the ICC chain as from the original pending intent sender to the final component that is started by the recipient, e.g. $a \rightarrow c$ in the previous example. The ICC chain information can also be passed and maintained using the intents, with modifications on both *ActivityManager* and the pending intent class.

Content Providers are Android components that do not rely on the intent mechanism for ICC. Rather, public APIs are directly used to access the Content Provider. To pass ICC chains to a Content Provider, the corresponding Content Provider APIs are intercepted. We also modified the *ContentResolver* class in the application context to enable the *ResourceManager* to pass ICC chains.

Passing to Resource Providers. In UPPDROID, we define system services (e.g. *LocationManager*), content providers (e.g. contacts) and kernel resources (e.g. *sdcard*) as three types of *resource providers*. When the recipient in an ICC is accessing these resource providers on behalf of the sender, the correct ICC chains need to be passed to determine how the resources are to be accessed. We describe how ICC chains are passed to system services and kernel resources, since passing to content providers are discussed above.

System services guarded by Android permissions are managed by various service managers at the framework level. In order to access these resources, apps first retrieve the service managers from the application context. Then, they invoke the APIs offered in these managers to access the resources. For example, location service is managed by the *LocationManager* which offers APIs such as *getLastKnownLocation*. As shown in Figure 5.4, we modify the application context to pass the ICC chains in the intent to these service managers in step 3. The APIs in

the service managers are also modified so that the ICC chains can factor in how to access the resources.

Kernel resources such as file system and network sockets are managed by the underlying Linux kernel. Our *Kernel ResourceManager* manages resource accesses at the kernel level. It maintains a separate copy of ICC chains with the help of *ActivityManager*, when an ICC occurs at the framework level. As shown in Figure 5.4, in step 4, right after *ActivityManager* appends the ICC chains in the intent i_2, it makes a system call to *Kernel ResourceManager* to pass the ICC chain.

5.3.2 Virtualizing Resource Accesses

UPPDROID uses a novel form of resource virtualization which is different from most existing uses of virtualization [24, 80, 82, 122]. In order to support app usability while preventing privilege escalation, we aim to design the system to be similar to what the app expects in the original Android. Our resource virtualization has the following features:

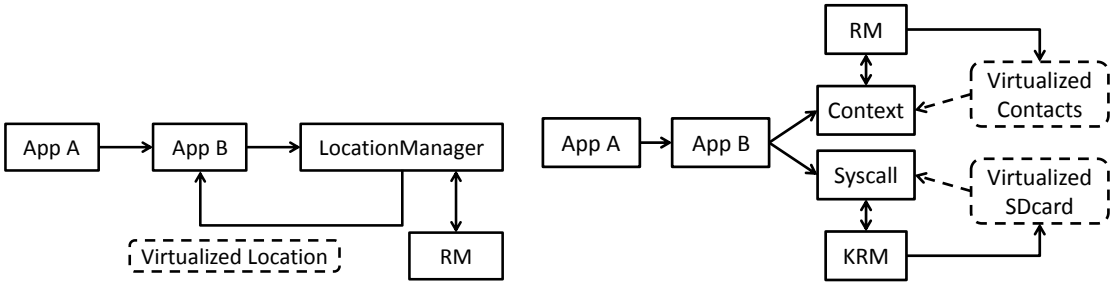
- *dynamic*: the resources are virtualized dynamically based on the ICC chains discussed in the previous rules. Rather than a system wide virtualization, the resources are virtualized on demand in a lightweight fashion. For example, rather than fully virtualize the whole *sdcard*, we can only virtualize a directory based on the ICC chain.
- *consistent*: the virtualized resources are uniquely associated with an ICC chain. We maintain the operations on the virtualized resources transparently so that in all intents with the same ICC, the previous resource is being used. We call this *resource consistency* in the same spirit as consistency in databases. For example, there can be two different virtualized *sdcard*, associated with $A \rightarrow C$ and $B \rightarrow C$ respectively. Next time the ICCs happen, A and B can each access the associated *sdcard*, which provides a consistent environment with past ICC usage. For example, in the virtual environment associated with $A \rightarrow C$, file f_A could be created initially and then subsequent intents could write to f_A but f_A would not be visible in the virtual environment associated with $B \rightarrow C$.

- *flexible*: the resource virtualization can be customized to trade off security with usability. First, the ICC chain used in the virtualization can be changed. Rather than using the entire ICC chain, we can use part of the chain, e.g. we can apply the intersections of permissions on the last two components along a particular ICC chain. By doing so, we relax the prevention of permission re-delegation allowing some re-delegation to enhance the usability or compatibility of the ICC with specific apps. Second, the resource can be virtualized in many ways. The virtualized resource can range from having an arbitrary value to one partially based on the real value to the real value itself which can have noise applied, e.g. the location resource can be a fake one or a coarse-grained value derived from the actual location with some noise added. The virtualized file system (or database) can be an exact shadow copy of the real file system (or database), a partial copy, or a copy with the same metadata without any actual data, or be just empty. The access on the file system can also be customized. For example, write access can be downgraded to read access or even no access. These customization can provide some usability in certain cases, like the coarse-grained values.
- *transparent*: the resource virtualization is transparent to both sender and recipient. The modifications on the virtualized resources do not affect the real resources. The recipient in an ICC can access the virtualized resources seamlessly.

We group Android resources into two categories, simple resources and environment resources, shown in Table 5.1. Simple resources are usually system services, which provide either a function or a result. They are accessed by either gaining the functionality or acquiring the result, e.g. gaining network connections and acquiring GPS locations. Environment resources are content providers and the file system, which provide operations, such as updating the database and modifying a document, on data and files. Not all the resources can be effectively virtualized, e.g. some simple resources do not return a value as results, such as camera and Internet. Either the accessing app gets the functionality or it does not. As it is not feasible to virtualize them, we deny the permissions to prevent privilege escalation as in the denial-based mechanisms.

Table 5.1: Category of System Resources

Simple Resource			Environment Resource	
Activity	Window	Layout_inflater	Applications	Calender
Alarm	Power	Notification	Contacts	Download
Keyguard	Search	Sensor	Media	PartnerBookmarks
Location	Storage	Vibrator	Telephony	Themes
Audio	Connectivity	WiFi	UserDictionary	SDcard
Media_router	Telephony	Input_method		
UI_mode	Download			



(a) Accessing Simple Resources (b) Accessing Environment Resources

Figure 5.5: Virtualizing System Resources

Virtualizing Simple Resources. For simple resources, we virtualize the values with several choices. The virtualized value can be a fake value (i.e. an artificially invented value which can be pre-defined or created based on the semantics of the simple resource) or a modification of the original value with some random noise. In the second case, the amount of noise can be specified for the ICC chain. A special case, is just a coarsening of the original value (can be thought of as the coarser value is obtained through the noise adjustment). These values are uniquely generated based on properties of the ICC chain, e.g. using ICC chain as hashes to generate virtualized values or noise, so that the recipient from the same ICC chain can always access the same value or approximate value (in the case of noisy values). Figure 5.5a shows how App B (with *ACCESS_FINE_LOCATION* permission) accesses location services on behalf of App A (without location permission) using virtualization. When App B invokes APIs in the *LocationManager*, the *LocationManager* queries the *ResourceManager*. Based on the ICC chain, the *ResourceManager* instructs the *LocationManager* to return a virtualized location, e.g. a fixed fake location.

Virtualizing Environment Resources. We virtualize the environment resource by creating a shadow database or a shadow file system respectively. Each shadow database or file system is uniquely generated and associated with an ICC chain identifier. The shadow database or file system can be empty or a partial copy or a full copy of the real resource, depending on the desired security policy and guarantee. By default, UPPDROID uses the empty copy to provide the greatest security against permission re-delegation. Since the shadow copy is maintained with the ICC chain, the recipient with the same ICC chain can access the copy in a consistent fashion.

Figure 5.5b shows how App B with *WRITE_EXTERNAL_STORAGE* and *READ_CONTACTS* permissions is accessing the contact list and *sdcard* on behalf of App A without any permission. When App B requests the contact list, the application context interacts with the *ResourceManager*. The *ResourceManager* generates a shadow contact list, e.g. an empty list, with a unique identifier of the ICC chain and instructs the application context to redirect the access to virtualized contact list. When App B accesses the *sdcard* through system calls, the system calls are intercepted by the *Kernel ResourceManager*. As ICC chains are passed to the *Kernel ResourceManager* by the *ActivityManager*, the *Kernel ResourceManager* creates a shadow *sdcard*, e.g. an empty *sdcard*, and redirects the system calls to the virtualized *sdcard*.

5.3.3 Implementation

We implement UPPDROID on Android 4.4. To support passing additional data through intent, we extended the original intent class to support new fields. These data are transparent to Android apps but available to the framework modules, such as *ActivityManager*. The pending intent class is also modified to support the ICC chain passing. We copy the data in a separate place once they reach the new Android component in an Android ICC. To pass data to Content Provider, we intercepted primary methods that handle the access to the Content Provider in application context, e.g., *query, insert, update, delete, getType*, etc. To generate and pass ICC chains in other Android components, we modified APIs, such as *startActivity, bindService, sendBroadcast*, with respect to each kind of components in *ActivityManager*. To pass ICC chains to system resources, we modified both the *getSystemServices* API in the applica-

Table 5.2: Resources evaluated with test app on UPPDROID

Resources	Functionalities
Microphone	record sound, media recording services
UserAccount	view and retrieve user accounts
SMS	send and receive SMS
PhoneCall	make telephone call
Camera	capture images
PhoneState	view various phone info, e.g device id
Location	get locations services
SDcard	access files on *sdcard*
Contacts	view and edit contacts on device
Other Contents	access other resources, e.g., call log, browser history

tion context and each system service manager, so that the system services can accept the ICC chains. We added a new system service *ResourceManager* to manage the resource virtualization at the framework level. We intercepted each service manager APIs accessing the resources, so that they cooperate with the *ResourceManager* to provide virtualization on the system resources. We modified the Content Provider class, so that the *ResourceManager* can generate new empty databases.

We added a kernel driver as the *Kernel ResourceManager* to manage the file system. To pass ICC chains to the *Kernel ResourceManager*, we provide new system calls to the driver. At the framework level, the *ActivityManager* uses these system calls to update the ICC chains to the *Kernel ResourceManager*. We created private folders on the real *sdcard* accessible to the *Kernel ResourceManager* to act as a shadow file system. We intercepted system calls that are related to file system access, such as *open, mkdir*, to support file system virtualization. They are redirected to access the virtualized file system.

5.4 Experimental Evaluation

We evaluated UPPDROID in a number of ways. Our evaluation was conducted on Android 4.4 KitKat. We first applied basic tests on resources using our own test app on the resources given in Table 5.2 to confirm that UPPDROID provides the expected functionality. We then tested with real app case studies, popular Android apps and performance benchmarks.

5.4.1 Case Studies

We now use three case studies to show why existing denial-based solutions result in loss of functionality or usability, for example, lead to an app crash. Our solution, on the other hand, attempts to preserve usability but still provides security through the use of resource virtualization. Naturally, there is no usability issue in the original Android since permission re-delegation is always allowed, however, there may be security breach due to the privilege escalation.

The resources tested in the case studies include file system (*sdcard*), system service (location) and content provider (contacts). These resources are guarded by the most frequently used dangerous permissions identified in the study by Wei et al. [121], such as *WRITE_EXTERNAL_STORAGE*, *ACCESS_FINE_LOCATION* and *READ_CONTACTS*. In each case, UPPDROID preserves app usability while preventing the privilege escalation problem, whereas denying the escalated permission does not.

We used a driver app (Myapptest) to start and interact with the tested apps in the case studies. This driver app can be configured to have various permission sets and request services from the tested apps. We run the tests in three environments:

- original Android – a clean Android 4.4;
- Android enhanced to deny elevated permissions through permission re-delegation, which we call a permission reduction approach [57] – Android 4.4 modified to simulate a permission reduction approach by reducing the permissions and applying the intersection of permissions in the request handling app when ICC happens; and
- UPPDROID– Android 4.4 with our modifications for UPPDROID.

File Manager App. *ES File Explorer File Manager* (`com.estrongs.android.pop`) is a file manager app which supports various file management functionalities, such as viewing and editing files and directories on the device. The normal scenario for using this file manager is to let an app request the file manager to view and edit the files in the *sdcard*.

In our test scenario, the driver app does not have the *WRITE_EXTERNAL_STORAGE* permission. It requests to edit a file `test.txt` in the *sdcard* using the file manager app. In

original Android, the driver app can read and write the file `test.txt` causing permission re-delegation.

In a permission reduction simulation, the *WRITE_EXTERNAL_STORAGE* permission is removed, resulting in the file manager app becoming unstable. The recipient first displays the *sdcard* to be empty, due to the lack of the *WRITE_EXTERNAL_STORAGE* permission, which has been removed to prevent privilege escalation. Any subsequent interactions on the file manager crash the app in unexpected ways without throwing any system exception messages. Sometimes, it can also crash immediately after being started by the driver app. By examining the system logs, we can see that there is a java security exception from a file system call *open* needing the *WRITE_EXTERNAL_STORAGE* permission. The result shows that denying the *WRITE_EXTERNAL_STORAGE* permission causes problems for the ES file manager app, which loses its functionalities servicing other apps on the system. For example, first we use another app B (We give these apps tags to help reading) to interact with this file manager app C to create directories and edit text files. Then the driver app A interacts with app C, causing a switch to a new activity of app C. When this new activity crashes due to the permission removal, the whole app C crashes and app B loses its unsaved data interacting with another activity of this file manager app C.

In UPPDROID, based on the rules we defined in the Section 5.3, *sdcard* is virtualized. The file manager app is redirected to access the virtual *sdcard*. For this test case, UPPDROID creates a virtualized *sdcard* where there are empty copies of files from the real *sdcard*.[2] The file `test.txt` is an empty file. When the file manager writes to the file, UPPDROID saves the updated copy to a private folder associated with the ICC chain $myapptest \rightarrow com.estrongs.android.pop$. Thus, there is no data leak from the file `test.txt`. When the file manager edits the virtualized file, the changes do not affect the original `test.txt`. Similarly, we use another app B to interact with the file manager app C. Then, we use the driver app A to interact with app C. In this case, the interaction between A and C does not affect the interaction between B and C. The data is kept separately from each other. The file manager app C does not crash and the driver app A does not gain any extra privileges from this ICC.

[2] The copies are lazily created when needed. Note that having shadow files is a configuration option in UPPDROID.

GPS Location. *GPS Coordinates GPS Location* (`gps.coordinates.share.sms.email`) is a location service app which offers basic location services, such as providing coordinates of the current location, and sharing the coordinates through SMS or emails. It requires the *ACCESS_FINE_LOCATION* permission. The normal usage of this GPS service app is to return GPS coordinates.

We configure the driver app to remove its *ACCESS_FINE_LOCATION* permission. The driver app is run to interact with the GPS service app to show the current GPS coordinates. Original Android allows this interaction, providing the current coordinates (37.422005, -122.084095) (simulated values set on the device for testing).

In the permission reduction simulation, the location service app crashes with a security exception showing that the permission is needed for the proper operation. In fact, exception handling shows a pop-up window, indicating the GPS-coordinates have stopped. While we only used a test driver app, a real app may be using another app providing GPS and other services in a number of ways. If the GPS service app crashes, it could cause significant impairment of functionality.

In UPPDROID, the location resource is virtualized based on the call chain and the virtualization rule, $R(P(GPSserviceapp) - P(Mytestapp) \cap P(GPSserviceapp))$. The virtual coordinates have been set as $(0, 0)$, though other options are also possible, e.g., a coarse-grained location. The result is that the GPS service app can still function with fake coordinates. With a completely fake value, there is no privilege escalation of the location resource in the driver app.

Phone Book Contacts. *Phone Book ConTacTs* (`com.brainworks.contacts`) is a phone book app which provides basic services to view and edit contacts and call logs. The *READ_CONTACTS* and *READ_CALL_LOG* permissions are required to access contacts and call logs.

We configure the driver app to remove its *READ_CONTACTS* permission. In original Android, it can start the contacts app freely. The phone contacts can be edited freely in the contacts app afterwards.

In the permission reduction simulation, the contacts app crashes when the *READ_CONTACTS* permission is revoked. The Android system catches this security exception, as the per-

mission is not presented when accessing the contacts content provider on device. A pop-up window shows that the ConTacTs app has stopped.

In UPPDROID, the contact list is virtualized according to the rule $R(P(PhoneBookConTacTs) - P(Mytestapp) \cap P(PhoneBookConTacTs))$. An empty contact list is created and provided to the Phone Book ConTacTs app. The contacts app shows the empty contact list on behalf of the driver app. Alternatively, we can also make it show fake contacts. Again there is no privilege escalation since the original resource access is prevented, but now the contacts app works properly without any exceptions.

UPPDROID works properly with the three virtualized resources from both simple and environment types. In fact, there are more than 250 similar apps[3] from each of their resource types which can suffer from this kind of problem when interacting with another app. Most of these apps require *WRITE_EXTERNAL_STORAGE*, *READ_CONTACTS* and *ACCESS_FINE_LOCATION* permissions to function properly.

5.4.2 Usability Improvement over Permission Restriction

To evaluate the usability improved by UPPDROID, we select 60 popular apps (20 for each category) that are location-related, contacts-related, storage-related apps. To compare the permission reduction simulation and our solution, we run these apps where the corresponding permissions are disabled. All of them crash due to the missing permissions when we tested using the *monkey tool*[4] and manual tests. We then ran them in UPPDROID with the exact same set of UI events in both automatic and manual tests. As expected, the apps no longer crash. UPPDROID supports the primary functionality for all the tested apps and correctly creates virtual resources for permission re-delegation in ICCs. Compared to the simple permission reduction simulation which leads to crashes in all tested apps, thus, making all these apps unusable, we see that UPPDROID significantly improves the user experience while preventing privilege escalation in the Android platform.

[3] *Google Play* only shows the most related 250 apps when searching for the three types of the apps. The actual number can be more.

[4] The tool that runs on the emulator generates pseudo-random streams of user events such as clicks, touches, or gestures, also a number of system-level events.

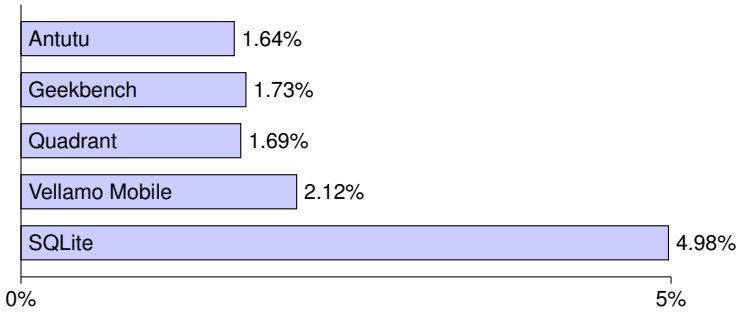


Figure 5.6: Runtime Overhead of UPPDROID

5.4.3 Performance Benchmarks

We used five of the most popular benchmarks, *Antutu*, *Geekbench*, *Quadrant Standard*, *Vellamo Mobile* and *SQLite* on *Google Play* to benchmark UPPDROID. The benchmarks were run with our prototype implementation on a Nexus 7 WiFi 2012 Android 4.4 device. The benchmarks test CPU, memory usage, I/O, and database accesses in Android. We use these benchmarks to evaluate the impact of UPPDROID on a real Android device under typical usage scenarios exercised by the benchmarks. The benchmarks do not specifically focus on ICC but still do exercise ICCs. As our prototype implementation involves changes at the Android framework and the Linux kernel level, these benchmarks also exercise these aspects. It is not feasible to test the numerous apps available on Android and many apps are infeasible to benchmark as they require GUI interaction.

We compare UPPDROID on Android 4.4 KitKat against original Android 4.4. Figure 5.6 gives the additional overhead relative to the original Android for the various benchmarks which is either the run-time or a benchmark score. The runtime overhead for UPPDROID is quite small for all the benchmarks. In the SQLite benchmark, which focuses on the database operations in the Content Provider, it is below 5%. In all other benchmarks it is below 2.2% which is quite negligible. Our Content Provider implementation currently does not rely on the general ICC chain passing mechanism; rather, it utilizes various hooks in the Content Provider APIs themselves. We expect that the small increase in overhead in the SQLite benchmark is because it focuses on Content Provider I/O which exercises our hooks more. However, real apps are usually less database intensive.

5.5 Related Work

Permission Re-delegation Detection and Mitigation. ComDroid [42], Woodpecker [65], CHEX [85] and SEFA [123] apply similar static analyses, e.g., reachability analysis, to detect in-component, cross-component, and cross-app vulnerabilities. However, static approaches may not identify all the possible permission re-delegation at the framework level. Prevention mechanisms are needed against privilege escalation from permission re-delegation. Quire [51] provides ICC provenance on Android to prevent a confused deputy attack (permission re-delegation). It tracks the call chain of IPCs, which allows reducing the privileges of the callee and acting on the callee's behalf. Scippa [27] provides a system-centric IPC provenance to Android. It modified both the framework and the kernel level IPC handling system. It provides a new system API for the apps to query the callers. UPPDROID provides a lightweight ICC provenance, where the ICC information is kept away from the apps. The usage of ICC provenance is to aid the virtualization of resources by not limiting the permissions themselves. IPC inspection [57] deals with permission re-delegation by reducing the permissions of the recipients. When dealing with the singleton apps, it completely revoke the permissions of the recipients, which can result in crashes affecting the usability of these apps. UPPDROID preserves the usability of the recipients with the proper security guarantee in the Android ICC to prevent permission re-delegation. Bugiel et al. [36] extend Xmandroid [35] to detect confused deputy and collusion attacks by monitoring the runtime ICCs. It performs framework level monitoring and enforces kernel level mandatory access control. However, it relies on the policies specified by users and designers to block the interactions without proper permissions. It does not focus the usability problem as UPPDROID does.

Permission System Enhancement. Apex [93] allows a user to authorize partial permissions and revoke granted permissions on demand. Saint [97] examines the interfaces one app exported to another and extends the Android framework to enforce inter-app security policy (at the install time and runtime). ConUCON [28] enforces usage control based on system context (e.g., device location). COMPAC [118] provides a fine-grained access control at the java component level and enables the developers to assign permissions to the components within apps. DNADroid [45] enforces control on both install-time permission assignment and runtime

permission usage. However, these approaches require the users to have a good understanding of permissions as well as policies and focus on a single app. In contrast, UPPDROID focuses on Android ICC which does not require user interactions but can integrate these policy-based approaches to extend its functionality.

Resource Virtualization. On the desktop platforms, resource virtualization ideas [84, 112, 129] are widely used to provide isolation. On the mobile platforms, L4Android [80] utilizes a micro-kernel to support multiple independent Android system virtual machines. However, the overhead of L4Android renders it impractical for mobile devices with limited resources. Cells [24] supports multiple virtual phone (VP) environment through light-weight OS virtualization. Users can switch through these VPs. AirBag [122] uses a light-weight OS virtualization to isolate untrusted mobile apps, with a restricted environment to run untrusted Android apps. RVDroid [82] provides virtualization by virtualizing resources to different virtual views, thus, confining apps to those views. UPPDROID builds on the primitives provided by RVDroid. Unlike RVDroid, which uses a full system virtualization into virtual views, we enhance Android ICC with a novel lightweight dynamic virtualization whose context is based on ICC chains and provides virtual resource consistency. While these approaches focus on regulating a single app, UPPDROID focuses on solving the permission re-delegation problem between multiple apps.

Data and Resource Protection. TISSA [132] implements a *privacy mode* on Android allowing users to control the access to personal information at runtime. It uses fake data to block the untrusted app from stealing the private information. TaintDroid [53] employs dynamic information flow tracking technique to detect whether sensitive data has been sent out through the network interfaces. AppFence [67] builds upon TaintDroid to prevent data leakage through network APIs. It applies data shadowing on primitive data with fixed values and content providers with an empty set. MockDroid [32] detects undesired permission requests and returns a resource reported as empty or unavailable if the permission is blocked by the user at run-time. Although most of these solutions employ the idea of using fake data, they do not virtualize the resources in a comprehensive way as UPPDROID, and focus data protection of individual app.

5.6 Summary

In this chapter, we propose a solution to prevent privilege escalation attacks on Android caused by permission re-delegation in the Android inter-component communication. Compared to existing solutions which are primarily based on reducing permissions to prevent the privilege escalation, we instead virtualize the resources. This still prevents privilege escalation but allows for better usability since operations in the recipients are not prevented nor do they fail. We also allow a larger range of security options by relaxing how virtualization is done. We show in case studies that indeed usability is enhanced as the permission denial approach causes app crashes while we do not. Our evaluation also shows that our solution has little overhead.

Chapter 6

Conclusion

Software components are designed for code reuse. They are the foundation of modern computer systems that encapsulate many sets of software functionalities. The complexity of these systems, together with the interactions among the software components and various system settings, can affect the software components and result in a component misuse vulnerability that alters the original functionalities provided by the software components, leading to security problems.

In this thesis, we aim to detect and prevent the misuse of software components. Motivated by a large amount of exploits in the wild and the limited security provided by existing solutions, we propose a number of solutions to detect the component misuse vulnerability and provide enhanced protection into the system.

We study how the misuse of components is caused by various system settings in complex computer systems. We develop a detection mechanism of binary loading vulnerabilities, which combines data dependency analysis with system dependency knowledge to identify the factors affecting the loading behaviors of the programs. We propose a loading dependency graph to explain binary loading by showing the dependencies from loading behaviors to the factors. It detects the unexpected uses of the loading behaviors based on the threat model discussed. We evaluate the approach with real world programs and identify 24 binary loading vulnerabilities. Our approach precisely identifies the vulnerable binaries.

We study how the misuse of components is caused by component interactions in complex

computer systems. We develop a detection and prevention mechanism of ActiveX API misuse vulnerabilities in Internet Explorer, using a reachability analysis. It provides an access control based mitigation solution to block vulnerable ActiveX methods in the browser programs. Our approach is able to identify misuses of APIs caused by component interactions in six real-world ActiveX controls. It also provides good efficiency with low overhead. With this approach, we prevent the API misuse from complex component interactions.

After studying the causes of the component misuse vulnerability, we propose a solution to mitigate the damage caused by the misuse in the interactions of software components. We develop a mechanism to mitigate the privilege escalation problem in Android ICC. It prevents the permission re-delegation in Android ICC that can allow the sender to gain additional privileges from the recipient. With the help of resource virtualization, the usability of this prevention is enhanced. We conduct case studies with real-world Android apps and perform performance evaluation with various benchmarks. The approach is proved to be effective and has a low overhead.

With these solutions, we detect the component misuse vulnerability in a systematic way and prevent the component misuse vulnerability with enhanced usability in complex computer systems.

6.1 Future Work

Enhancing Dynamic Tracking Mechanisms. The dynamic tracing mechanism introduced in detecting misuses of ActiveX controls focuses on tracking the function calls in the programs. We plan to track various forms of data flows as we did in detecting binary loading vulnerabilities. With current heuristics, we can already identify how the IE ActiveX blocking mechanism is bypassed by leveraging the library loading APIs, discussed in Section 4.4.2.

The program tracing mechanism to detect binary loading vulnerabilities is a prototype implementation, which directly logs all the instructions in the virtual machine. It generates instruction traces with a size of more than 10 gigabytes for a single program to initialize itself. We plan to reduce the size of the traces in the future. For example, we can remove certain

information for a simple instruction, which is not involved in the data dependency propagation. The trace format can also be optimized to further shrink the size of the trace.

Enriching the Test Cases. The dynamic tracking mechanisms rely on test cases. More test cases can help in identifying misuses in software components if they contain the misused behaviors. We plan to support more form of test case generations in the ActiveX API misuse tracking mechanism. Currently, the test cases are only generated for JavaScript in the IE browser. We plan to support more scripts, such as VBScript, and more Windows programs, such as Office programs.

For the test cases in the binary loading detection, we plan to make the testing of configuration file "smarter". We plan to create heuristics from various orthogonal analyses [46, 130] to find configuration options controlling the binary loading behaviors. LDRSCOPE is designed for the component designers, who can use their own test cases in the software components.

Extending the Prevention Mechanisms. The current prevention mechanism of API misuse vulnerabilities in ActiveX controls discussed in Section 4.4.3 is based on blacklisting. We plan to extend the mechanism by adding more customizations and still preserve its fine-grained control over the ActiveX methods. The extension can involve using security policies and creating various whitelists based on system knowledge.

For prevention mechanism of privilege escalation problem in Android ICC, we discussed the customization of security levels in Section 5.3. We plan to integrate security policies [36,97] discussed in orthogonal solutions into our approach, so that the Android ICC can be monitored and secured in a fine-grained manner.

Bibliography

[1] http://www.oreillynet.com/onlamp/blog/2008/05/safari_carpet_bomb.html.

[2] http://developer.android.com/guide/components/fundamentals.html.

[3] http://technet.microsoft.com/en-us/security/advisory/2269637.

[4] http://secunia.com/advisories/windows_insecure_library_loading/.

[5] http://blog.acrossecurity.com/2011/07/binary-planting-goes-any-file-type.html.

[6] http://http://msdn.microsoft.com/en-us/library/windows/desktop/ff951640(v=vs.85).aspx.

[7] http://en.wikipedia.org/wiki/DLL_injection.

[8] http://www.microsoft.com/security/resources/activex-whatis.aspx.

[9] http://technet.microsoft.com/en-us/security/dn535768.aspx.

[10] http://windows.microsoft.com/en-sg/windows/security-zones-adding-removing-websites#1TC=windows-7.

[11] http://msdn.microsoft.com/en-us/library/windows/desktop/aa367061(v=vs.85).aspx.

[12] https://technet.microsoft.com/en-us/library/security/ms06-014.aspx.

[13] http://research.microsoft.com/en-us/projects/detours/.

[14] http://www.securityfocus.com/bid/10514.

[15] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0003.

[16] http://support.microsoft.com/kb/955617.

[17] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2463.

[18] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5002.

[19] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5660.

[20] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2007-5660.

[21] http://www.exploit-db.com/exploits/4050/.

[22] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing memory error exploits with wit. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 263–277. IEEE, 2008.

[23] P. Ammann, D. Wijesekera, and S. Kaushik. Scalable, graph-based network vulnerability analysis. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 217–224. ACM, 2002.

[24] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 173–187. ACM, 2011.

[25] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: analyzing the android permission specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228. ACM, 2012.

[26] Todd M Austin and Gurindar S Sohi. Dynamic dependency analysis of ordinary programs. In *ACM SIGARCH Computer Architecture News*, volume 20, pages 342–351. ACM, 1992.

[27] Michael Backes, Sven Bugiel, and Sebastian Gerling. Scippa: system-centric ipc provenance on android. In *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[28] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-aware usage control for android. In *Proceedings of Security and Privacy in Communication Networks*, pages 326–343, 2010.

[29] David Barrera, William Enck, and Paul C Van Oorschot. Meteor: Seeding a security-enhancing infrastructure for multi-market application ecosystems. In *IEEE MoST: Mobile Security Technologies Workshop*, 2012.

[30] David Barrera, H Güneş Kayacik, Paul C van Oorschot, and Anil Somayaji. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM, 2010.

[31] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell. State of the art: Automated black-box web application vulnerability testing. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 332–345. IEEE, 2010.

[32] Alastair R Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, pages 49–54. ACM, 2011.

[33] Theodore Book, Adam Pridgen, and Dan S. Wallach. Longitudinal analysis of android ad library permissions. *Computing Research Repository*, abs/1303.0857, 2013.

[34] David Brumley, Dawn Xiaodong Song, Tzi-cker Chiueh, Rob Johnson, and Huijia Lin. RICH: automatically protecting against integer-based vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2007, San Diego, California, USA, 28th February - 2nd March 2007*, 2007.

[35] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technical Report TR-2011-04, Technische Universität Darmstadt, 2011.

[36] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.R. Sadeghi, and B. Shastry. Towards taming privilege-escalation attacks on android. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[37] C. Cadar, D. Dunbar, and D. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 209–224, 2008.

[38] C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. Exe: automatically generating inputs of death. *ACM Transactions on Information and System Security*, 12(2):10, 2008.

[39] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: triage for market-scale mobile malware analysis. In *Proceedings of the sixth ACM conference on Security and privacy in wireless and mobile networks*, pages 13–24. ACM, 2013.

[40] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of ACM Conf. on Computer and Communications Security*, pages 235–244, 2002.

[41] Kevin Zhijie Chen, Noah M Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Thomas R Magrino, Edward XueJun Wu, Martin Rinard, and Dawn Xiaodong Song. Contextual policy enforcement in android applications with permission event graphs. In *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[42] E. Chin, A.P. Felt, K. Greenwood, and D. Wagner. Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM, 2011.

[43] M. Christodorescu, S. Jha, S.A. Seshia, D. Song, and R.E. Bryant. Semantics-aware malware detection. In *Proceedings of Symposium on Security and Privacy*, pages 32–46. IEEE, 2005.

[44] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *Proceedings of 22nd Annual Computer Security Applications Conference*, pages 269–278. IEEE, 2006.

[45] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proceedings of 17th European Symposium on Research in Computer Security*, pages 37–54, 2012.

[46] W. Cui, M. Peinado, K. Chen, H.J. Wang, and L. Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.

[47] H. Dai, C. Murphy, and G. Kaiser. Configuration fuzzing for software vulnerability detection. In *Proceedings of ARES'10 International Conference on Availability, Reliability, and Security*, pages 525–530. IEEE, 2010.

[48] V Danen. CVE-2011-1658: ld.so ORIGIN expansion combined with RPATH. `https://bugzilla.redhat.com/showbug.cgi?id=CVE-2011-1658`.

[49] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy. Privilege escalation attacks on android. *Information Security*, pages 346–360, 2011.

[50] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. Mocfi: A framework to mitigate control-flow attacks on smartphones. In *Proceedings of Network and Distributed System Security Symposium*, 2012.

[51] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D.S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[52] W. Dormann and D. Plakosh. Vulnerability Detection in ActiveX Controls through Automated Fuzz Testing, 2008. `http://www.cert.org/archive/pdf/dranzer.pdf`.

[53] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*, pages 393–407, 2010.

[54] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[55] Michael D Ernst. Static and dynamic analysis: Synergy and duality. In *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003.

[56] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[57] Adrienne Porter Felt, Helen J Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[58] V. Ganapathy, S. Jha, D. Chandler, D. Melski, and D. Vitek. Buffer overrun detection using linear programming and static analysis. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 345–354. ACM, 2003.

[59] V. Ganesh, T. Leek, and M. Rinard. Taint-based directed whitebox fuzzing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 474–484. IEEE, 2009.

[60] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 1–1. ACM, 2007.

[61] P. Godefroid, A. Kiezun, and M.Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of ACM SIGPLAN Notices*, volume 43, pages 206–215. ACM, 2008.

[62] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *Proceedings of ACM Sigplan Notices*, volume 40, pages 213–223. ACM, 2005.

[63] P. Godefroid, M.Y. Levin, D. Molnar, et al. Automated whitebox fuzz testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2008.

[64] Patrice Godefroid and Nils Klarlund. Software model checking: Searching for computations in the abstract or the concrete. In *Integrated Formal Methods*, 2005.

[65] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[66] Bhargav S Gulavani, Thomas A Henzinger, Yamini Kannan, Aditya V Nori, and Sriram K Rajamani. Synergy: a new algorithm for property checking. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006.

[67] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. 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*, pages 639–652. ACM, 2011.

[68] Y.W. Huang, S.K. Huang, T.P. Lin, and C.H. Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proceedings of the 12th international conference on World Wide Web*, pages 148–159. New York, NY, USA, 2003.

[69] K. Ingols, M. Chu, R. Lippmann, S. Webster, and S. Boyer. Modeling modern network attacks and countermeasures using attack graphs. In *Proceedings of Computer Security Applications Conference*, pages 117–126. IEEE, 2009.

[70] Michael Jang. *Linux Annoyances for Geeks: Getting the Most Flexible System in the World Just the Way You Want It.* " O'Reilly Media, Inc.", 2006.

[71] S. Jha, O. Sheyner, and J. Wing. Two Formal Analyses of Attack Graphs. In *Proceedings of IEEE Computer Security Foundations Workshop*, pages 49–63, 2002.

[72] L. Juranić. Using fuzzing to detect security vulnerabilities. *Technical report, Infingo IS*, 2006.

[73] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, 2007.

[74] Kristen Kennedy, Eric Gustafson, and Hao Chen. Quantifying the effects of removing permissions from android applications. In *Proceedings of the Workshop on Mobile Security Technologies (MoST)*, 2013.

[75] S.T. King and P.M. Chen. Backtracking Intrusions. *ACM Transactions on Computer Systems*, 23(1):51–76, 2005.

[76] T. Kwon and Z. Su. Automatic detection of unsafe component loadings. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 107–118. ACM, 2010.

[77] T. Kwon and Z. Su. Automatic detection of unsafe dynamic component loadings. *IEEE Transactions on Software Engineering*, 38(2):293–313, 2012.

[78] T. Kwon and Z. Su. Static detection of unsafe component loadings. *Compiler Construction*, pages 122–143, 2012.

[79] Carl E Landwehr, Alan R Bull, John P McDermott, and William S Choi. A taxonomy of computer program security flaws. *ACM Computing Surveys (CSUR)*, 26(3):211–254, 1994.

[80] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 39–50. ACM, 2011.

[81] D. Larochelle, D. Evans, et al. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, volume 10, 2001.

[82] Xiaolei Li, Guangdong Bai, Zhenkai Liang, and Heng Yin. A software environment for confining malicious android applications via resource virtualization. In *Proceedings of the 18th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2013.

[83] Donglin Liang and Mary Jean Harrold. Slicing objects using system dependence graphs. In *Proceedings of the International Conference on Software Maintenance*, pages 358–367. IEEE, 1998.

[84] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the 19th Annual Computer Security Applications Conference*, pages 182–191. IEEE, 2003.

[85] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM, 2012.

[86] C.K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, 2005.

[87] Pratyusa K Manadhata and Jeannette M Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, 2011.

[88] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. Mitchell. A Layered Architecture for Detecting Malicious Behaviors. In *Proceedings of Recent Advances in Intrusion Detection*, pages 78–97. Springer, 2008. LNCS 5230.

[89] Charlie Miller and Zachary NJ Peterson. Analysis of mutation and generation-based fuzzing. *White Paper, Independent Security Evaluators, Baltimore, Maryland (securityevaluators.com/files/papers/analysisfuzzing.pdf)*, 2007.

[90] D. Molnar, X.C. Li, and D.A. Wagner. Dynamic test generation to find integer bugs in x86 binary linux programs. In *Proceedings of the 18th conference on USENIX security symposium*, pages 67–82. USENIX Association, 2009.

[91] A. Morais, E. Martins, A. Cavalli, and W. Jimenez. Security protocol testing using attack trees. In *Proceedings of International Conference on Computational Science and Engineering*, volume 2, pages 690–697. IEEE, 2009.

[92] A. Moser, C. Kruegel, and E. Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 231–245. IEEE, 2007.

[93] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings*

of the 5th ACM Symposium on Information, Computer and Communications Security, pages 328–332. ACM, 2010.

[94] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[95] S. Noel and S. Jajodia. Managing attack graph complexity through visual hierarchical aggregation. In *Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*, pages 109–118. ACM, 2004.

[96] S. Noel, S. Jajodia, B. O'Berry, and M. Jacobs. Efficient minimum-cost network hardening via exploit dependency graphs. In *Proceedings of Computer Security Applications Conference*, pages 86–95. IEEE, 2003.

[97] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[98] T Ormandy. CVE-2010-3847: GNU C library dynamic linker $ORIGIN expansion vulnerability. `http://www.exploit-db.com/exploits/15274/`.

[99] T Ormandy. CVE-2011-1658: ld.so ORIGIN expansion combined with RPATH. `http://www.exploit-db.com/exploits/15304/`.

[100] X. Ou, W.F. Boyer, and M.A. McQueen. A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345. ACM, 2006.

[101] M. Payer, T. Hartmann, and T.R. Gross. Safe loading-a foundation for secure execution of untrusted programs. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 18–32. IEEE, 2012.

[102] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Proceedings of the 7th*

ACM Symposium on Information, Computer and Communications Security, pages 71–72. ACM, 2012.

[103] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.

[104] 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 Network and Distributed System Security Symposium*, 2014.

[105] Filippo Ricca and Paolo Tonella. Construction of the system dependence graph for web application slicing. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, pages 123–132. IEEE, 2002.

[106] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[107] F.B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

[108] O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. Automated generation and analysis of attack graphs. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[109] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the International Conference on Software Engineering*, pages 432–441. IEEE, 1999.

[110] C. Song, J. Zhuge, X. Han, and Z. Ye. Preventing Drive-by Download via Inter-Module Communication Monitoring. In *Proceedings of ACM Symposium on Information, Computer and Communications Security*, pages 124–134, 2010.

[111] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena. Bitblaze: A new approach to computer security via binary analysis. *Information systems security*, pages 1–25, 2008.

[112] Weiqing Sun, Zhenkai Liang, VN Venkatakrishnan, and R Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.

[113] M. Sutton and A. Greene. The art of file format fuzzing. In *Proceedings of Blackhat USA Conference*, 2005.

[114] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[115] H.H. Thompson, J.A. Whittaker, and F.E. Mottay. Software security vulnerability testing in hostile environments. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 260–264. ACM, 2002.

[116] Timothy Vidas, Nicolas Christin, and Lorrie Cranor. Curbing android permission creep. In *Proceedings of the 2011 Web 2.0 Security and Privacy Workshop*, volume 2, 2011.

[117] T. Wang, T. Wei, G. Gu, and W. Zou. Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *Proceedings of IEEE Symposium on Security and Privacy (SP)*, pages 497–512. IEEE, 2010.

[118] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. Compac: Enforce component-level access control in android. In *Proceedings of the 4th ACM conference on Data and application security and privacy*. ACM, 2014.

[119] Y.M. Wang, D. Beck, B. Vo, R. Roussev, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 368–377. IEEE, 2005.

[120] Richard Wartell, Yan Zhou, Kevin W Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *Machine Learning and Knowledge Discovery in Databases*. 2011.

[121] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Permission evolution in the android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[122] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium*, 2014.

[123] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 623–634. ACM, 2013.

[124] Y. Wu and R.H.C. Yap. Towards a binary integrity system for windows. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 503–507. ACM, 2011.

[125] Y. Wu, R.H.C. Yap, and R. Ramnath. Comprehending Module Dependencies and Sharing. In *Proceedings of ACM/IEEE International Conference on Software Engineering*, pages 89–98, 2010.

[126] Yongzheng Wu, Sai Sathyanarayan, Roland H. C. Yap, and Zhenkai Liang. Codejail: Application-transparent isolation of libraries with tight program interactions. In *Proceedings of the 17th European Symposium on Research in Computer Security*, pages 859–876, 2012.

[127] Z. Wu, J.W. Atwood, and X. Zhu. A new fuzzing technique for software vulnerability mining. *In Proceedings of the IEEE CONSEG 09: International Conference on Software Engineering*, 9, 2009.

[128] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis.

In *Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[129] Yang Yu, Fanglu Guo, Susanta Nanda, Lap-chung Lam, and Tzi-cker Chiueh. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 24–34. ACM, 2006.

[130] Jiaqi Zhang, Lakshminarayanan Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. Encore: exploiting system environment and correlation information for misconfiguration detection. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*, pages 687–700. ACM, 2014.

[131] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 611–622. ACM, 2013.

[132] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.