Taeho Kwon
December 2011
Computer Science

Detecting and Analyzing Insecure Component Integration

Abstract

Component technologies have been widely adopted for designing and engineering software applications and systems, which dynamically integrate software components to achieve desired functionalities. Engineering software in a component-based style has significant benefits, such as improved programmer productivity and software reliability. To support component integration, operating systems allow an application to dynamically load and use a component. Although developers have frequently utilized such a system-level mechanism, programming errors can lead to insecure component integration and serious security vulnerabilities. The security and reliability impact of component integration has not yet been much explored.

This dissertation systematically investigates security issues in dynamic component integration and their impact on software security. On the conceptual level, we formulate two types of insecure component integration—*unsafe component loading* and *insecure component usage*—and present practical, scalable techniques to detect and analyze them. Our techniques operate directly on software binaries and do not require source code. On the practical level, we have used them to discover new vulnerabilities in popular, real-world software, and show that insecure component integration is prevalent and can be exploited by attackers to subvert important software and systems. Our research has had substantial practical impact and helped to mitigate unsafe component loadings on Microsoft Windows applications.

Detecting and Analyzing Insecure Component Integration

By

TAEHO KWON

B.S. (Handong Global University) 2001
M.S. (Korea Advanced Institute of Science and Technology) 2003

DISSERTATION

Submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

Zhendong Su, Chair

Premkumar Devanbu

Hao Chen

Committee in Charge

2011

To my dearest wife Jihee and adorable son Youngmin.

Contents

List of Figures

List of Tables

Abstract

Component technologies have been widely adopted for designing and engineering software applications and systems, which dynamically integrate software components to achieve desired functionalities. Engineering software in a component-based style has significant benefits, such as improved programmer productivity and software reliability. To support component integration, operating systems allow an application to dynamically load and use a component. Although developers have frequently utilized such a system-level mechanism, programming errors can lead to insecure component integration and serious security vulnerabilities. The security and reliability impact of component integration has not yet been much explored.

This dissertation systematically investigates security issues in dynamic component integration and their impact on software security. On the conceptual level, we formulate two types of insecure component integration—*unsafe component loading* and *insecure component usage*—and present practical, scalable techniques to detect and analyze them. Our techniques operate directly on software binaries and do not require source code. On the practical level, we have used them to discover new vulnerabilities in popular, real-world software, and show that insecure component integration is prevalent and can be exploited by attackers to subvert important software and systems. Our research has had substantial practical impact and helped to mitigate unsafe component loadings on Microsoft Windows applications.

Acknowledgments

Many people have supported my graduate studies in Davis. I'm certain that their supports have helped my research a lot. I would like to express my sincere appreciation on them.

My most sincere thanks go to my advisor Professor Zhendong Su. He has fully supported my research and endeavored to educate me as an independent researcher. Besides research skills, I have learned positive attitude and passion while working with him. I really appreciate him from the bottom of my heart. I believe that what I have learned from him will be a foundation for my future life.

Jihee, my dearest wife, has lived as a firm supporter of me and a mother of our adorable son Youngmin during my graduate studies. I truly appreciate her effort to take care of our family, making me keep motivated for my research. Without her sincere help, I may not be able to focus on my research. I can never thank her enough. Also, my parents and parents-in-law have contributed to my graduate studies in a variety of ways. I am grateful to their help.

I would like to express my thanks to my colleagues: Earl Barr, Lingxiao Jiang, Mark Gabel, Andreas Saebjoernsen, Fanqi Sun, Liang Xu, Zhongxian Gu, Vu Minh Le, Thanh Vo, and Mehrdad Afshari. Their sincere effort for novel research has made a positive influence on me as a researcher during my graduate studies. Also, their constructive comments have provided me with insight to help improve my research. I really appreciate their help and hope to keep in touch with them.

Last but not least, it is a particular pleasure to thank Jesus Christ, who has made right decisions in my life. Also, I appreciate the members of Davis Korean Church for their encouragements and fellowship during my graduate studies.

Chapter 1

Introduction

Component-based software development has been a major paradigm for engineering software. It allows developers to seamlessly reuse well engineered and tested software components. The benefits of component-based development do come with a price—unsafe programming can lead to insecure component integration and introduce security vulnerabilities. However, the security and reliability issues involving component integration have not been much explored. The goal of this dissertation is to systematically investigate these issues and how they impact software security.

In this chapter, we introduce component-based software development and describe the core of component integration. We then formulate the concept of insecure component integration and present relevant real-life security vulnerabilities. Finally, we survey previous research on component security and discuss the structure of the rest of the dissertation.

1.1 Component-based Software

The size and complexity of modern software system have been significantly increasing [103]. For example, the line of code (LOC) for Microsoft Windows NT products has been increased about ten folds from 1993 to 2003. In particular, Windows NT 3.1 is about 5 million LOC, while Windows Server 2003 is about 50 million LOC [138]. The increased software size and complexity have led to several important challenges of software development. First, they led to significantly increased effort for software development. Second, software also becomes much more expensive to maintain [16]. Third, the increased software size and complexity provide additional surface for

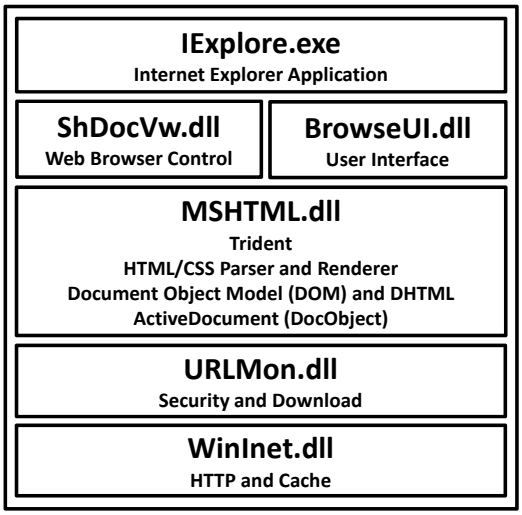


Figure 1.1: Internet Explorer architecture [78].

reliability and security concerns [103].

To mitigate these challenges, component-based software development has been widely adopted. In particular, software is built from multiple components and operates by utilizing their functionalities. This paradigm has made software development more productive and reliable. First, developers can reuse components with designed functionalities to avoid writing redundant code. Second, components may be developed in parallel because they usually serve as independent software modules. Third, programs using reusable components may be more reliable [9]. Because of these advantages, component-based development has been widely adopted to develop many real-world applications. As an example, Figure 1.1 shows the architecture of Internet Explorer, which is composed of several components and operates by invoking relevant functions of these components.

1.2 Component Integration

The essence of component-based software development is its support for integrating components. To this end, modern operating systems typically support *dynamic loading* [47]. Figure 1.2 shows a high-level overview of this typical mechanism with three phases: *resolution*, *loading*, and *usage*. Suppose that component `A` needs to invoke a function `foo` exported by component `B`. In this case, the operating system first determines the fullpath of component `B` (resolution phase) and incorporates the component `B` into the client software's memory address space (loading phase). Later component

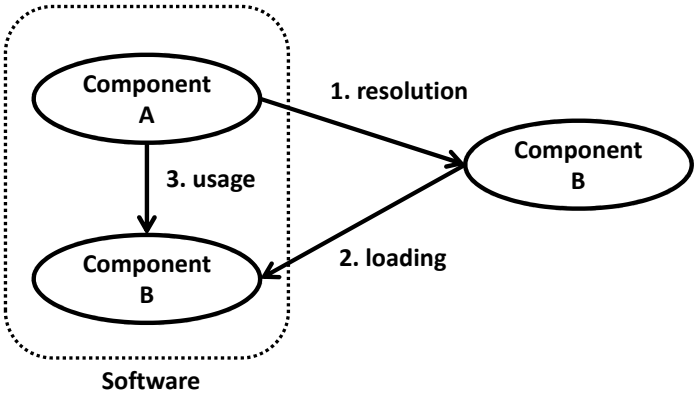


Figure 1.2: Component integration.

A invokes `foo` of the loaded component `B` (usage phase).

A client can request component resolution and loading at either loadtime or runtime. If component `A` calls `foo` through a standard API (*e.g.*, `foo()`), it implicitly resolves and loads component `B` at loadtime. In this case, when invoking `foo` at runtime, component `A` executes the code located at the memory address corresponding to the entry point of `foo`. In this mechanism, even though `foo` is not invoked at runtime, component `B` always resides in the memory space of the client, component `A`. To mitigate this issue, operating systems support APIs that resolve and load components, and retrieve the addresses of the entry points of the desired functions at runtime. For example, Microsoft Windows provides system APIs, `LoadLibrary` and `GetProcAddress`, to support component resolution and loading.

1.3 Insecure Component Integration

Component integration allows developers to effectively reuse components such as system and third-party libraries at runtime. For component reuse, developers need (1) to specify the component implicitly or explicitly, and (2) to determine how to use the loaded component. To this end, developers generally specify the component in terms of its filename and acquire knowledge on its secure usage from the documentation. This standard development process can suffer from inherent security concerns. First, malicious components with the specified filename can be loaded instead of the intended one, because operating systems resolve the target component based on its filename. Second, the documentation on a component may not be sufficient to fully understand how to securely

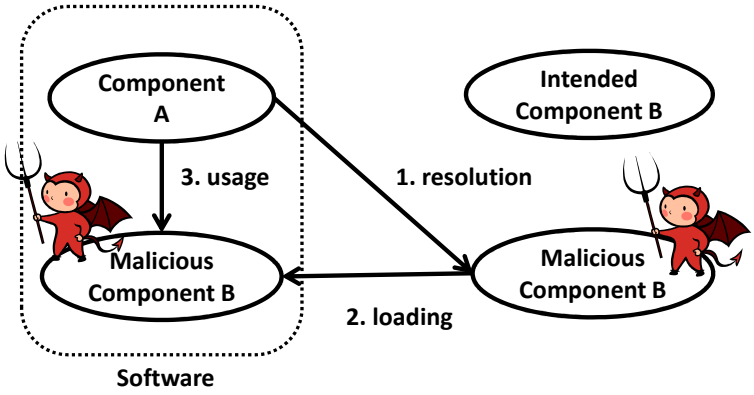


Figure 1.3: Unsafe component loading.

use the component, or even worse, documentation may not even be available. Thus, it is easy for developers to misuse a component, leading to security vulnerabilities.

These concerns can lead to real-life security threats. In particular, unsafe programming can cause insecure component integration, which can be misused by attackers to subvert software systems. This section introduces two types of insecure component integration, *unsafe component loading* and *insecure component usage*, and presents several real-life attack vectors.

1.3.1 Unsafe Component Loading

For runtime safety and security, an application should only load its intended components. To this end, it is necessary for the resolution phase to locate the correct component for use at runtime. Operating systems generally provide two resolution methods, either specifying the *fullpath* or the *filename* of the target component. With fullpath, operating systems simply locate the target from the given full path. With filename, operating systems resolve the target by searching a sequence of directories, determined by the runtime *directory search order*, to find the first occurrence of the component.

Although flexible, this common component resolution strategy has an inherent security problem. Since only a file name is given, unintended or even malicious files with the same file name can be resolved instead. Figure 1.3 illustrates such unsafe loading and its impact. Suppose that developers intend component `A` to resolve component `B`. In this case, attackers can hijack the loading of component `B` by placing malicious component named `B` in a directory searched before the directory where the intended component resides. When the malicious component is loaded, the attackers can

execute arbitrary code.

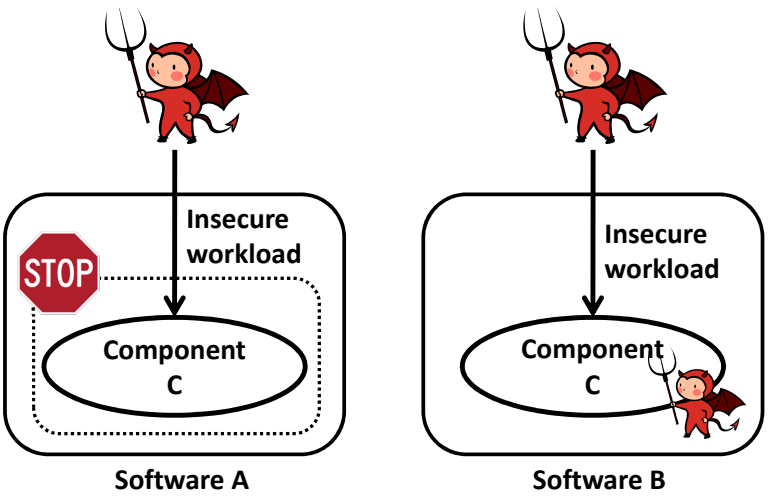
This problem had been known for a while, but it had not been considered a serious threat because its exploitation requires local file system access on the victim host. Recently, the problem has started to receive more attention. In particular, our study shows that unsafe loadings on Microsoft Windows are prevalent and can lead to remote code execution attacks [94]. Remote attacks are possible for two main reasons: 1) the OS looks for a component with a given file name and cannot distinguish malicious ones from benign ones with the same file name; and 2) the default directory search order on Microsoft Windows contains the current directory (*i.e.*, "."), where remote attackers can trick a victim user to download files to via social engineering or by exploiting other vulnerabilities.

Here is an example attack scenario on Microsoft Windows. An attacker sends a victim user via email an archive that contains an arbitrary `.asx` file and a malicious file named `rapi.dll`. The user extracts the archive file and runs `Winamp 5.58` to open the `.asx` file, the `rapi.dll` is loaded, which leads to a remote code execution attack [94]. Besides archive files, the Carpet-Bomb attack [114] and the WebDAV protocol [6] can be exploited for launching remote attacks. This very issue has also received considerable recent media coverage [50,107,111,125,160]. Microsoft released MS10-087, rated "Critical," to patch Microsoft Office [152]. To mitigate the issue, Microsoft also released a fix-it tool to control the directory search order by introducing a new registry key [77,108]. However, it changes the default system-wide setting and leads to backward compatibility issues. Fundamentally, this is a safe programming issue, and Microsoft provides programming guidelines for safe dynamic loading [46] and is conducting an ongoing investigation to secure the loading procedure [105].

1.3.2 Insecure Component Usage

Software is commonly built from reusable components that provide desired functions. Although component reuse significantly improves software productivity, insecure component usage can lead to security vulnerabilities in client applications. Figure 1.4 depicts an example of insecure component usage. Suppose that (1) software `A` and `B` reuse component `C` to execute insecure workloads such as HTML rendering, and (2) the expert knowledge on the usage of component `C` is only available for developing software `A`. In this case, software `A` can be protected from malicious attacks based on the insecure workloads, software `B` can be vulnerable to such attacks.

As real-life examples, we noticed that common IE-based browsers, such as IE Tab, disable



(a) Blocked insecure workload. (b) Bypassed insecure workload.

Figure 1.4: Insecure component Usage.

important security features that IE enables by default, even though they all use the same browser components. First, IE enables an XSS filter by default [71]. However, IE-based browsers use the same browser components as IE, but do not enable the XSS filter. This insecure component usage makes these IE-based browsers vulnerable to XSS attacks. Second, while IE enables `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE`, IE-based browsers do not. The configuration of this security policy is checked by both IE and IE-based browsers at runtime, but the inconsistent configuration data lead them to behave differently. Specifically, IE-based browsers allow user names and passwords in URL address, leading to potential attack vectors for phishing [79].

1.4 Previous Research

As discussed in Section 1.3.2, insecure component integration can lead to security vulnerabilities in client software. However, this class of security issues has not yet been systematically investigated in the literature. Instead, prior research has focused on security problems in a single component: (1) analyzing vulnerabilities in software components, or (2) detecting malicious components.

1.4.1 Defect Detection in Components

Viewing each component as an independent piece of software, this line of work focuses on detecting vulnerabilities in individual components. For example, many effective techniques based on concolic

testing [55, 131] have been developed to test various types of software, such as C programs [27, 29, 55, 102, 131], binary executables [56, 128], and Web applications [7, 155].

The tools [10, 11, 36, 38, 44] for detecting vulnerabilities in ActiveX components have been developed in industry. Each tool adopts software testing techniques such as fault injection to generate malicious inputs triggering errors in the components.

To predict which components may be vulnerable, Neuhaus *et al.* [117] propose to statistically analyze the vulnerability history; function calls and imports of each vulnerable component are utilized to characterize the corresponding vulnerabilities.

1.4.2 Malicious Component Detection

Detecting malicious software components has been mainly performed for (1) behavioral malware clustering, (2) malware signature generation/matching, and (3) behavioral malware detection.

Behavior-based Malware Clustering Lee and Mody [97] model the behavior of a target process with its event sequence and define the similarity metric as transform cost between two sequences, which is based on the Levenshtein distance. Bailey *et al.* [12] utilize the Backtracker system [86] to generate profiles about system state changes by target malware samples and apply a machine learning algorithm to cluster the profiles by using normalized compression distance. Rieck *et al.* [126] collect behavior reports of target malware by using *CWSandbox* [65] and utilize the support vector machine technique to classify numerical vectors determined by string analysis. Bayer *et al.* [22] perform advanced dynamic analysis based on information of system call traces, taint data and network traffic, build behavior profiles based on OS objects and the set of all operations accessing them, and present an efficient LSH-based hierarchical clustering algorithm based on the Jaccard index as a similarity metric.

Malware Signature Generation/Matching Christodorescu *et al.* [33] construct, from dynamic traces, a directed acyclic graph (DAG) where each node corresponds to a collected system call and an edge to a dependency among the system calls. Kolbitsch *et al.* [89] improve this DAG-based behavior model by considering dataflow dependency determined by system call arguments. MetaAware [167] analyzes a sequence of system calls from disassembled malware binaries through static control and data flow analysis. Sathyanarayan *et al.* [151] extract a signature of particular

malware family by computing the mean frequency of critical API calls from malware binaries. Kruegel *et al.* [91] proposes malware variant detection based on identifying common subgraphs in two control flow graphs. This approach shows high detection rate (97 out of 100) on worm instances in a single malware family. Christodorescu *et al.* [34] presents semantic-aware malware detector using templates, which specify malicious behavior based on instruction sequences with variables and symbolic constants. To detect malware, they developed two templates of loop decryption and Massmailer and matched them existing in three kinds of malware variants.

Behavioral Malware Detection Yin *et al.* [164] propose the notion of taint graphs to represent taint propagation from a source and utilized it to detect malware. For malware detection, they specify policies defining invalid accesses to introduced inputs and analyze taint graphs on introduced inputs to check whether or not potential malware samples violate pre-defined policies. Types of invalid accesses the authors utilize include anomalous information access, anomalous information leakage, and excessive information access. Kirda *et al.* [87] describes behavioral invariant of spyware of BHO and toolbar types. According to the paper, the spyware component leaks information on user behavior monitored by interacting with the web browser. Based on this behavioral invariant, they presented a spyware detection technique that can find browser COM functions and Windows API calls executed in response to simulated browser events through a combination of dynamic and static analysis. Kruegel *et al.* [93] proposes a technique to detect rootkits through static analysis locating instruction sequences for data transfer for illegal memory access and a write operation using kernel-level address calculated by a forbidden kernel symbol reference, resulting in no false positives/negatives. To detect drive-by download attacks usually exploited by spyware, Moshchuk *et al.* [113] define five trigger conditions and monitor them during web browsing: process creation, file system activity, suspicious processes that write to a file, registry activity and browser or OS crash. Each condition specifies suspicious behavior that is not supposed to occur during normal web-page rendering. Based on these conditions, a large set of spyware can be detected.

1.5 Dissertation Structure

This dissertation aims at understanding insecure component integration and analyzing its effect on software security. We structure the remainder of this dissertation as follows:

Formalizing insecure component integration Chapter 2 formulates insecure component integration by presenting formal definitions of unsafe component loading and insecure component usage.

Detecting unsafe component loadings Chapter 3 presents the first *dynamic analysis* for detecting unsafe component loadings and the evaluation results on their prevalence and severity on both Microsoft Windows and Linux. In particular, our results show that unsafe component loading is prevalent in software on both OS platforms, and it is a more severe concern for Microsoft Windows. Our tool detected more than 4,000 unsafe component loadings, some of which lead to remote code execution.

Although the technique in Chapter 3 is effective in detecting real security errors, it suffers from limited code coverage and may miss important vulnerabilities. Thus, it is desirable to develop effective techniques to detect all possible unsafe component loadings. To this end, Chapter 4 presents the first *static binary analysis* aiming at detecting all possible loading-related errors, and evaluates its effectiveness against the dynamic technique on popular Windows applications.

Detecting insecure component usage Chapter 5 presents the first practical framework for detecting and analyzing vulnerabilities of insecure component usage. We have implemented our technique for Windows applications and used it to detect and analyze insecure usage of popular software components. Our evaluation results show that our framework is scalable and effective at detecting and analyzing insecure usages. In particular, it enabled us to detect several serious, new vulnerabilities and helped perform detailed analysis of insecure component usage.

Conclusion Chapter 6 summarizes this dissertation and discusses future research directions.

Chapter 2

Formulation of Insecure Component Integration

In this chapter, we formulate two types of the insecure component integration, *unsafe component loading* and *insecure component usage*.

2.1 Unsafe Component Loading

This section describes dynamic loading of components and formulates its unsafety.

2.1.1 Dynamic Loading of Components

Software components often utilize functionalities exported by other components such as shared libraries at runtime. This operation is generally composed of three phases: *resolution*, *loading*, and *usage*. Specifically, an application resolves the needed target components, loads them, and utilizes the desired functions provided by them.

Component integration can be achieved through *dynamic loading* provided by operating systems or runtime environments. For example, the `LoadLibrary` and `dlopen` system calls are used for dynamic loading on Microsoft Windows and Unix-like operating systems respectively. Dynamic loading is generally done in two steps: *component resolution* and *chained component loading*.

Component resolution. In order to resolve a target component, it is necessary to specify it correctly. To this end, operating systems provide two types of target component specifications: *fullpath*

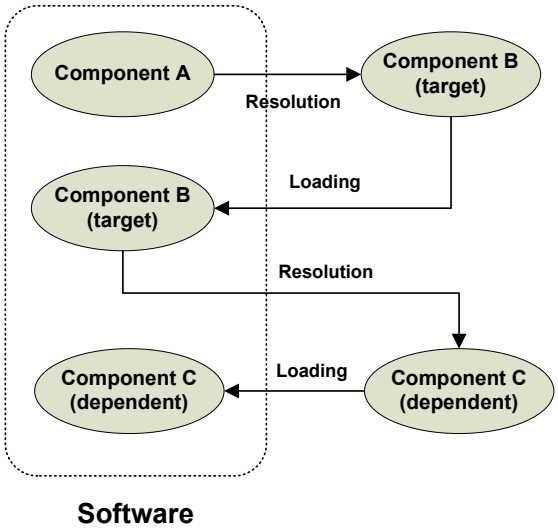


Figure 2.1: Dynamic component loading procedure.

and *filename*. For fullpath specification, operating systems resolve a target component based on the provided fullpath. For example, a fullpath specification `/lib/libc-2.7.so` for the libc library in Linux determines the target component using the specified full path. For filename specification, operating systems obtain the full path of the target component from the provided file name and a dynamically determined sequence of search directories. In particular, an operating system iterates through the directories until it finds a file with the specified file name, which is the resolved component. For example, suppose that a target component is specified as `midimap.dll` and the directory search order is given as `C:\Program Files\iTunes;C:\Windows\System32;...;$PATH` on Microsoft Windows. If the first directory containing a file with the name `midimap.dll` is `C:\Windows\System32`, the resolved full path is determined by this directory.

Chained component loading. In dynamic loading, the full path of the target component is determined by its specification through the resolution process, and the component is incorporated into the host software if it is not already loaded. During the process of incorporating the target component, the component's load-time dependent components are also loaded. Figure 2.1 illustrates the general procedure of dynamic loading. Suppose component B is loaded by component A. B's dependent components (*e.g.*, component C) are also loaded. We can usually obtain information on B's dependent components from B's file description. This process of chained component loading is repeated until all dependent components have been loaded.

2.1.2 Formal Definition

Dynamic component loading is commonly supported by operating systems through specific system calls that take as input a full path or file name for the intended component. For example, Microsoft Windows provides component-loading system calls such as `LoadLibraryA`. Once such a system call is invoked, the OS resolves the target component as follows:

- The target component can be specified by its full path or its file name.
- When full path is used, the OS directly resolves the target using the provided full path.
- Otherwise, if file name is used and known by the OS, the full path of the specified file is predefined. For example, `KERNEL32.DLL` is known by Microsoft Windows and its full path is predefined as `"C:\WINDOWS\SYSTEM32\KERNEL32.DLL"`.
- If the given file name is unknown to the OS, it iterates through the predefined search directories to locate the first file with the specified file name.

To formalize the component resolution process, it is necessary to model the *file system state*, because even the same component-loading code may result in different resolutions under different file system states. We define a file system state s to be the set of full paths of all files stored on the current file system.

Definition 2.1.1 (Component Resolution) *A component resolution function R takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \Sigma^*$ and a file system state s, and returns a resolved full path $\pi \in \Sigma^*$, where Σ denotes the alphabet used to specify files and directories.*

- *If f is a full path,*

$$R(f, d, s) = \begin{cases} f & \text{if } f \in s; \\ \epsilon & \text{otherwise}. \end{cases}$$

  where ϵ is the empty string.

- *If f is a file name,*

$$R(f,d,s) = \begin{cases} \pi & \textit{if } f \textit{ is known to the OS as } \pi; \\ d_k + \backslash + f & \textit{if } S = \{i \mid d_i + \backslash + f \in s\} \\ & \quad \wedge\ S \neq \emptyset\ \wedge\ k = \mathsf{min}(S); \\ \epsilon & \textit{otherwise}. \end{cases}$$

where "+" denotes string concatenation.

We next formalize component loading, for which we need to consider the currently loaded components. The reason is that the OS does not load the same component multiple times. In our formalization, we let Π denote the set of full paths of all the currently loaded components.

Definition 2.1.2 (Component Loading) *Given the loaded components Π, a component loading function L takes a component specification $f \in \Sigma^*$, a directory search order $d = \langle d_1, \ldots, d_n \rangle \in \Sigma^* \times \ldots \Sigma^*$, a file system state s, and the set of loaded components Π, and returns a resolution success or failure:*

$$L(f,d,s,\Pi) = \begin{cases} \textit{success} & \textit{if } R(f,d,s) \notin \{\epsilon\} \cup \Pi; \\ \textit{failure} & \textit{otherwise}. \end{cases}$$

The formalized component loading mechanism in Definition 2.1.2 is commonly used on major operating systems. However, as OS determines a target component only through its name, unsafe programming can make software load an unintended component with the same name. Attackers can exploit this security vulnerability by modifying the file system state. In particular, the loading of a target component can be hijacked if a malicious file with the same name can be created in a directory searched before the directory where the intended component resides. This component hijacking can be misused for local or remote attacks [94].

To formalize unsafe component loading, it is necessary to determine the current file system state as whether or not a component loading is safe is relative to a file system state. We first define a *normal file system state* w.r.t. an application p.

Definition 2.1.3 (Normal File System State) *A file system state s is* normal *w.r.t. an application p*

if no unintended components are loaded while p executes in state s. We use s_p to denote a normal file system state w.r.t. the application p.

We formalize two types of unsafe loadings: *resolution failure* and *unsafe resolution*. We use R_p and L_p to denote component resolution and component loading performed by an application p, respectively.

Definition 2.1.4 (Resolution Failure) *For an application p, a resolution failure occurs at runtime if $R_p(f, d, s_p) = \epsilon$. In this case, with a full path specification f, an arbitrary file with the same full path f can hijack the component loading. If f is file name, an attacker can hijack this loading by placing a file (or tricking the user to place a file) with the specified name f in any* writable *directory d_i by the attacker under the search order $d = \langle d_1, \ldots, d_n \rangle$.*

Definition 2.1.5 (Unsafe Resolution) *For an application p, an unsafe resolution occurs at runtime if the following conditions hold: 1) f is the file name of the target component and unknown to the OS; 2) $R_p(f, d, s_p) = d_k + \backslash + f \wedge \ k > 1$; and 3) $L_p(f, d, s_s, \Pi) = success$. In this case, an attacker can hijack the loading by placing a file (or tricking the user to place a file) with the specified name f in any* writable *directory d_i by the attacker where $i < k$.*

To avoid unsafe loadings, it is necessary for developers to specify the target component in a safe manner. We define safe target component specifications as follows.

Definition 2.1.6 (Safe Component Spec) *Under a given threat model, a loading specification for an application p is safe if either of the following holds: 1) if f is a full path, $R(f, d, s_p) \neq \epsilon$ and the attacker cannot overwrite f or trick the user to overwrite f; and 2) if f is an unknown file name to the OS, $R(f, d, s_p) = d_i + \backslash + f$ and the attacker cannot place a file or trick the user to place a file named f in any of the d_j for $1 \leq j \leq i$.*

2.1.3 Dynamic Loading-related Remote Attacks

As we mentioned in Section 2.1.2, insecure component resolutions may cause an application to load unintended components. This issue had been known for a long time, but it had not been considered a serious threat because it requires local file system access on the victim host for exploitation.

Recently, realistic attacks exploiting vulnerable component loading have been discovered, including ones by us. In this section, we describe these attack vectors.

"Carpet Bomb"-based Attack

The *Carpet Bomb* attack [1] can lead to remote code execution in conjunction with unsafe DLL loading on Microsoft Windows. In particular, when the Safari browser accesses a malicious web page, attackers can make the browser automatically download arbitrary files to the user's Desktop directory without any prompting. This is referred to as the Carpet Bomb attack. This flaw leads to remote code execution if a vulnerable application checks in the Desktop directory first for resolving a DLL. For example, suppose `sqmapi.dll` is downloaded onto the victim's Desktop directory through the Carpet Bomb attack. When Internet Explorer 7 runs, it loads this DLL file and executes arbitrary code [75]. Microsoft released software patches [109, 110] to fix this vulnerability.

"Shortcut with Component" Attack

Sending a victim an archive file containing a shortcut to a vulnerable program and a malicious component can also cause remote code execution. If the vulnerable program starts up via the shortcut, it loads the component and executes malicious code.

This flaw can be exploited through social engineering-based attacks. For example, Foxit PDF Reader 4.2 running via its shortcut will load `peerdist.dll` placed in the same directory as the shortcut on Microsoft Windows Vista. Attackers can deceive the victim to run Foxit Reader through its shortcut to access interesting PDF documents. This way they can exploit this vulnerability by making the PDF reader load the provided malicious DLL.

Furthermore, this attack vector can be combined with the Carpet Bomb attack. Because shortcuts tend to be placed in the Desktop directory, running a vulnerable application such as Foxit Reader via its shortcut can load the relevant components stored on the Desktop through the Carpet Bomb attack.

"Document with Component" Attack

Opening a document can load particular files placed in the same directory as the document. This vulnerability can be exploited to launch remote code execution attacks by sending a victim an archive

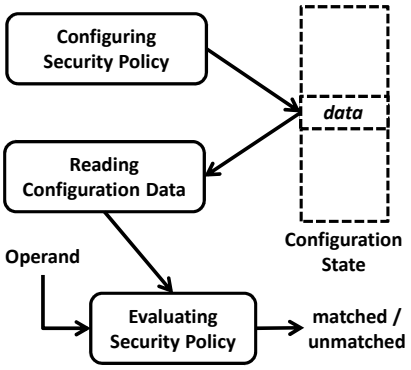


Figure 2.2: Security policy configuration and evaluation.

file containing a document and a malicious component.

For example, suppose that a user opens an arbitrary document for Microsoft Word 2010 on Windows 7. In this case, `IMESHARE.dll`, located in the same directory as the document, is loaded when the program runs. This flaw can lead to serious security threats in Microsoft Word 2010.

2.2 Insecure Component Usage

This section formalizes insecure component usage.

2.2.1 Security Policy-related Execution

A security policy configuration serves as a key part of software protection because it determines whether or not certain malicious behavior is to be blocked (*e.g.*, IE XSS filter). Figure 2.2 depicts the runtime process of configuring and evaluating security policies: 1) software maintains its global state and updates the global state to configure a security policy; and 2) when evaluating the policy, the software reads data from the global state and checks whether or not the data match a specified operand. Based on the above description, we formally define security policy-related execution.

Definition 2.2.1 (Configuration State) *A* configuration state $M = [\langle k_1, v_1 \rangle, \ldots, \langle k_n, v_n \rangle]$ *is an associative array whose key and value pair* $\langle k_i, v_i \rangle$ *corresponds to a configured security policy identifier and its configuration data, respectively.*

We define the configuration and evaluation of a security policy in terms of accesses to a configuration state M.

Definition 2.2.2 (Security Policy Configuration) *For a given configuration state M, a security policy configuration function $conf$ updates M based on a new policy configuration $\langle k, v\rangle$ where k and v correspond to a policy identifier and its configuration data, respectively: $conf(M, k, v) = M'$ such that*

$$M'(k') = \begin{cases} v & \textit{if } k' = k \\ M(k') & \textit{otherwise} \end{cases}$$

Definition 2.2.3 (Security Policy Evaluation) *Given a configuration state M, a security policy evaluation function $eval$ takes k (a policy identifier) and p (a specified operand for its evaluation):*

$$eval(M, k, p) = \begin{cases} \textit{matched} & \textit{if } M[k] = p \\ \textit{unmatched} & \textit{otherwise} \end{cases}$$

Based on the above definitions, we next define security policy-related execution.

Definition 2.2.4 (Security Policy-related Execution) *A security policy-related execution for a workload w of a software S, denoted by $\pi(S, w)$, is a sequence of policy configurations or evaluations $\pi(S, w) = \langle s_1, \ldots, s_m\rangle$ where $s_i = conf(M, k, v)$ (*i.e.*, a policy configuration) or $s_i = eval(M, k, p)$ (*i.e.*, a policy evaluation).*

Note that in Definition 2.2.4, the configuration state M changes at runtime, because S dynamically configures new security policies during its execution over the workload w. Also note that $\pi(S, w)$ provides us with precise information on policy evaluations during S's execution over w. We now define security policy evaluation patterns.

Definition 2.2.5 (Security Policy Evaluation Pattern) *For a given $\pi(S, w)$, the security policy evaluation pattern $epat(\pi(S, w))$ is the sub-sequence $epat(\pi(S, w)) = [\langle k_i, p_i, eval(M, k_i, p_i)\rangle]$ extracted from the policy evaluations from $\pi(S, w)$ where k_i, p_i, and $eval(k_i, p_i)$ correspond to a*

policy identifier, an operand for its policy evaluation, and the evaluation result (either matched *or* unmatched*) respectively.*

2.2.2 Formal Definition

For a reference R and a test subject T, we define two types of inconsistent policy configurations: *missing* and *incorrect configurations*.

Definition 2.2.6 (Missing Configuration) *The test subject T misses the configuration of a policy k evaluated by the reference R if $\exists p, s(\langle k, p, s\rangle \in epat(\pi(R, w))$ and $\forall p, s(\langle k, p, s\rangle \notin epat(\pi(T, w))$.*

Definition 2.2.7 (Incorrect Configuration) *The test subject T incorrectly configures a policy k if $\exists p, r_1 \neq r_2(\langle k, p, r_1\rangle \in epat(\pi(R, w)) \wedge \langle k, p, r_2\rangle \in epat(\pi(T, w)))$.*

Inconsistent policy configurations can lead to unprotected software execution; we define it as follows.

Definition 2.2.8 (Unprotected Software Execution) *Suppose that a security policy k blocks a malicious behavior Φ at runtime. Given a reference R, a test subject T, and a common workload w, T is unprotected w.r.t. k if R blocks Φ but T does not.*

Suppose that a component C maintains its configuration state M at runtime. If a client using C configures a policy stored by M insecurely, it can be unprotected w.r.t. the policy. We next define insecure component usage.

Definition 2.2.9 (Insecure Component Usage) *Suppose that a component C maintains a security policy k that blocks a malicious behavior Φ at runtime. For a reference R and a test subject T that use C, T insecurely uses C w.r.t. k if T is unprotected w.r.t. k.*

Chapter 3

Dynamic Detection of Unsafe Component Loadings

3.1 Introduction

Dynamic loading is an important mechanism for software development. It allows an application the flexibility to dynamically link a component and use its exported functionalities. Its benefits include modularity and generic interfaces for third-party software such as plug-ins. It also helps to isolate software bugs as bug fixes of a shared library can be incorporated easily. Because of these advantages, dynamic loading is widely used in designing and implementing software.

A key step in dynamic loading is component resolution, *i.e.*, locating the correct component for use at runtime. Operating systems generally provide two resolution methods, either specifying the *fullpath* or the *filename* of the target component. With fullpath, operating systems simply locate the target from the given full path. With filename, operating systems resolve the target by searching a sequence of directories, determined by the runtime *directory search order*, to find the first occurrence of the component.

Although flexible, this common component resolution strategy has an inherent security problem. Since only a file name is given, unintended or even malicious files with the same file name can be resolved instead. Thus far this issue has not been adequately addressed. In particular, we show that unsafe component loading represents a common class of security vulnerabilities on the

Windows and Linux platforms (*cf.*, Section 3.4). Operating systems may provide mechanisms to protect system resources. For example, Microsoft Windows supports Windows Resource Protection (WRP) [3] to prevent system files from being replaced. However, these do not prevent loading of a malicious component located in a directory searched before the directory where the intended component resides.

The problem of unsafe dynamic loading had been known for a while, but it had not been considered a serious threat because its exploitation requires local file system access on the victim host. The problem has started to receive more attention due to recently discovered remote code execution attacks [50, 64, 94, 107, 125, 148, 160, 166]. Here is an example attack scenario. Suppose that an attacker sends a victim an archive file containing a document for a vulnerable program (*e.g.*, a Word document) and a malicious DLL. In this case, if the victim opens the document after extracting the archive file, the vulnerable program will load the malicious DLL, which leads to remote code execution (*cf.*, Section 2.1.3).

In this chapter, we present the first automated technique to detect unsafe dynamic component loadings. We cast our technique as a two-phase dynamic analysis. In the first phase, which is online, we use dynamic binary instrumentation to capture a program's sequence of events related to component loading (*dynamic profile generation*). In particular, we dynamically collect three kinds of information: 1) *system calls invoked for dynamic loading* for information on target component specifications, directory search orders, and the sequence of component loading behavior; 2) *image loading* for information on resolved component paths, and 3) *process and thread identifiers* for multi-threaded applications. In the second phase, which is offline, we analyze the captured profile to detect unsafe component resolutions (*offline profile analysis*). We detect two types of unsafe loadings—*resolution failure* and *unsafe resolution*—for each component loading from the profile. A resolution failure corresponds to the case where the target component is not found, while an unsafe resolution corresponds to the case where there exist other directories searched before the directory containing the resolved target component.

To evaluate our technique, we have implemented a set of tools for detecting unsafe component loadings on the Microsoft Windows family (*i.e.*, XP SP3, Vista SP2, and Windows 7) and Ubuntu 10.04 (a popular Linux distribution). We conducted an extensive analysis of the prevalence and severity of unsafe component loadings in popular software applications: 27 for Windows and 24 for

Linux. Our results show that unsafe component loading is prevalent on all analyzed platforms—we found 3,269 unsafe loadings on Windows and 752 on Ubuntu.

Our results also show that unsafe loadings on Windows cause more serious security concerns over those on Linux. In particular, the unsafe loadings, commonly detected in the Windows applications, can be exploited for local or remote attacks. There are two main reasons for this difference. First, most Windows users run with administrative privilege. Because of this insecurity, attackers can gain write permission to the directories where malicious files are created for hijacking component loadings. In comparison, Linux in general does not grant such permission to ordinary users. Second, Windows searches the current directory, *i.e.*".", for component loading by default. This Windows-specific mechanism can lead to remote execution attacks. For example, the current directory can be written by remote attackers via "Carpet Bomb" and social engineering-based attacks (*cf.*, Section 2.1.3). Indeed, we found 41 vulnerabilities (*i.e.*, 41/3,269) that can be easily exploited for remote code execution. Although Microsoft supports a mechanism to exclude the current directory from the directory search order, many Windows applications do not adopt it (*cf.*, Section 3.5.2). Besides these detected vulnerabilities, we describe additional remotely exploitable attacks in an earlier version of this chapter [94]. We reported the most serious vulnerabilities to Microsoft and collaborated with Microsoft engineers to address these issues.

We make the following main contributions:

- We present an effective dynamic analysis to detect vulnerable and unsafe dynamic component loadings. To our knowledge, this work introduces the first automated technique to detect and analyze vulnerabilities and errors related to dynamic component loading.

- We have realized our technique as a set of practical tools for detecting unsafe component loadings on Microsoft Windows and Linux. We have conducted an extensive analysis of unsafe component loadings on various types of popular software.

- We have discovered new remote attack vectors based on the findings from our analysis, which Microsoft confirmed and had actively worked with us and other software vendors to develop engineering solutions to patch. We also discuss and propose techniques to mitigate unsafe component loadings.

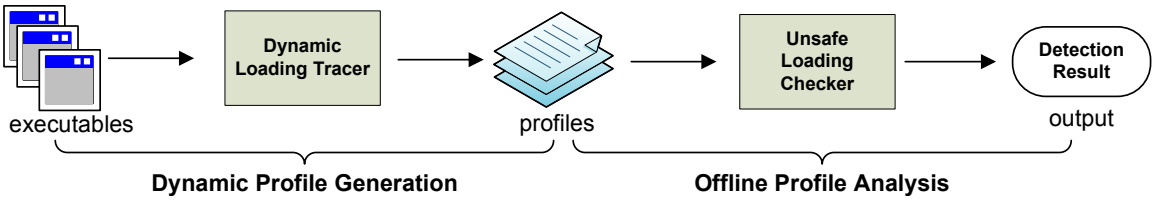


Figure 3.1: Framework for detecting unsafe component loadings.

The remainder of this chapter is structured as follows. In Section 3.2, we present our general technique to detect unsafe dynamic loadings. In Section 3.3, we describe background on dynamic component loading and implementation details of our tools for detecting unsafe component loadings on both Windows and Linux. Section 3.4 presents the evaluation of our tools, including characteristics and exploitability of the detected vulnerable and unsafe component loadings and our tools' performance. We also discuss techniques to mitigate unsafe component loadings (Section 3.5). Finally, we survey related work (Section 3.6) and conclude with a discussion of future work (Section 3.7).

3.2 Detection of Unsafe Loadings

As we mentioned in Section 2.1.3, unsafe component loading can cause serious security vulnerabilities in software. In this section, we present a dynamic analysis technique for detecting unsafe component loadings.

Figure 3.1 shows the high-level overview of our analysis process, which is composed of two phases: *dynamic profile generation* and *offline profile analysis*. To detect unsafe component resolutions, we first capture a sequence of system-level actions for dynamic loading during a program's execution. We use dynamic binary instrumentation to generate the profile on its runtime execution. We then reconstruct dynamic loading information from the profile offline and check safety conditions for each resolution. Because our technique only requires binary executables, it is robust and can be applied to analyze not only open source applications but also commercial off-the-shelf products.

Alternatively, we could also detect unsafe component loading during the program execution. However, we divide our analysis into two phases (*i.e.*, the dynamic profile generation and the offline profile analysis) to reduce the performance overhead incurred during dynamic binary instrumentation.

3.2.1 Dynamic Profile Generation

Dynamic analysis has been widely used to understand software behavior [39]. We also adopt this approach for detailed analysis of component loading. Specifically, we dynamically instrument the binary executable under analysis to capture a sequence of system-level actions for dynamic loading of components. During the instrumented program execution, we collect three types of information: *system calls invoked for dynamic loading*, *image loading*, and *process and thread identifiers*. The collected information is stored as a profile for the instrumented application and is analyzed in the offline profile analysis phase.

System calls invoked for dynamic loading. System call analysis is a widely used analysis technique to understand program behavior because a sequence of invoked system calls (with names of the invoked functions and their arguments) can provide useful information on program execution. To capture system-level actions for dynamic component loading, we instrument system calls that cover all possible control-flow paths of the dynamic loading procedure, which enables us to reconstruct the procedure offline.

Besides the name of an instrumented system call, we also collect its parameter information for detecting unsafe component resolutions. Specifically, the target component specification (*i.e.*, specified fullpath or filename) and the directory search order can be obtained from the system call parameters. Although the directory search order can vary according to the underlying system and program settings, it is computed by operating systems and provided as parameters to the relevant system calls for dynamic loading. Furthermore, results of the instrumented system calls provide both the control flow in the loading procedure and error messages generated by the operating systems. Such information is used for the reconstruction of the dynamic loading procedure and the detection of unsafe loadings.

Image loadings. We also capture actual loadings of target components via dynamic binary instrumentation. The loading information is needed for reconstructing the loading procedure in combination with the information captured by system call instrumentation. It also indicates the resolved full path determined by the loading procedure. We use this resolved path to detect unsafe component loading.

Process and thread identifiers. Because our approach is based on system call instrumentation, it

Type	Conditions
Resolution failure	1. Target component is not found
Unsafe resolution	1. Target component is specified by its name 2. Target component is resolved by iterating through multiple directories 3. There exists another searched directory before the resolution

Table 3.1: Conditions for detecting unsafe component loadings.

is important to consider multi-threaded applications. If the target program uses multi-threads and each thread loads a component dynamically, the instrumented system calls for each loading can be interleaved, which makes it difficult to correctly reconstruct the loading procedure of each thread. To solve this problem, we capture process and thread identifiers along with the other information on instrumented system calls. Note that dynamic binary instrumentation engines such as Pin [100] support API calls to capture the identifiers. With this additional information, we can analyze dynamic loadings of each thread by grouping its system calls using these recorded identifiers.

3.2.2 Offline Profile Analysis

In this phase, we extract each component loading from the profile and detect the unsafe loadings of a target component and its dependent components (*cf.* Section 2.1.1).

In the first step of this offline phase, we extract each component loading from the profile. To this end, we first group a sequence of actions in the profile by process and thread identifiers as the actions performed by different threads may be interleaved due to context switching. This grouping separates the sequences of dynamic loadings performed by different threads. Next, we divide the sequence for each thread into sub-sequences of actions, one for each distinct dynamic loading. This can be achieved by using the first invoked system call for dynamic loading (*e.g.*, `dlopen`) as a delimiter. After this step, we obtain a list of groups, each of which contains a sequence of actions for loading a component at runtime. This gives the possible control-flows in the dynamic loading procedure. Note that each group contains loading actions for both the target component and the load-time dependent components (*cf.* Section 2.1.1).

Our analysis detects the two types of unsafe component resolution that we discussed in Section 2.1.2: resolution failure and unsafe resolution. To this end, we check the conditions in Table 3.1, which are directly derived from the definition of each unsafe component resolution, for

Algorithm 1 OfflineProfileAnalysis

Input: S (a sequence of actions for a dynamic loading)
Auxiliary functions:
TargetSpec(S): return target specification of S
DirSearchOrder(S): return directory search order used in S
ImgLoad(S): return the image loadings in S
ResolutionFailure(S): return the resolution failures in S
ChainedLoading(S): return actions for the chained loadings in S
IsUnsafeResolution(filename, resolved_path, search_dirs): check whether the resolution is unsafe

```
1:  img_loads ← ImgLoad(S)
2:  failed_resolutions ← ResolutionFailure(S)
3:  if |img_loads| == 0 then
4:     if |failed_resolutions| == 1 then
5:        Report this loading as a resolution failure
6:     end if
7:  else
8:     spec ← TargetSpec(S)
9:     dirs ← DirSearchOrder(S)
10:    if spec is the filename specification then
11:       resolved_path ← img_loads[0].resolved_path
          // retrieve the first load
12:       if IsUnsafeResolution(spec,resolved_path,dirs) then
13:          Report this loading as an unsafe resolution
14:       end if
15:    end if
16:    chained_loads ← ChainedLoading(S)
17:    for each_load in chained_loads do
18:       OfflineProfileAnalysis (each_load)
19:    end for
20: end if
```

each component loading. Details of our offline profile analysis are given in Algorithm 1.

Resolution failure of a target component. To detect failed resolution of a target component, we simply check the number of image loads and the number of failed resolutions during the dynamic loading procedure. In particular, if no image is loaded and the resolution of the component failed, we report the component loading as a resolution failure (lines 3–6). Note that an OS does not load the same component multiple times. Thus, line 3 checks this necessary condition for resolution failure because a program may attempt to load a component that is already loaded. To avoid reporting any false resolution failures, we also explicitly check whether a resolution failure has occurred (line 4).

Unsafe resolution of a target component. Lines 10–15 describe how to detect unsafe resolution

of a target component. We first check whether the target component is specified by its file name, because a full path specification does not iterate through the search directories for resolution. If a file name is used, we retrieve the resolved path of the target component by retrieving the first element of a list of image loads in the dynamic loading procedure. Note that the first element of the list corresponds to the target component, because 1) there exists no image load in the loading procedure if the target component is already loaded or its resolution fails, and 2) the target component is always loaded for the first time during its runtime loading. Based on the resolved full path, the target component specification and the applied directory search order, we determine whether to classify this as an unsafe resolution by checking the directories searched before the resolution.

Unsafe component resolution by chained loadings. In lines 16–19, we detect unsafe component resolutions in the chained loading procedure by performing the offline profile analysis recursively. In particular, we extract each component loading from the chained loadings and recursively apply the aforementioned technique to detect unsafe component resolutions.

3.3 Implementation

To evaluate our proposed technique, we have developed tools to detect unsafe component loadings on the Microsoft Windows family (XP SP3, Vista SP2, and Windows 7) and Ubuntu 10.04, a popular Linux distribution. This section presents the implementation details of our tools.

3.3.1 Microsoft Windows Family

In this section, we provide background information on loading of Dynamic Link Libraries (DLLs) on Microsoft Windows and the implementation details of our tool for Windows 7. Details on our implementation for Windows XP and Vista can be found in an earlier version of this chapter [94].

Background on DLL Loading

Target DLL resolution. Microsoft Windows supports both types of target DLL specifications: fullpath and filename. For filename specifications, there exist Windows-specific mechanisms to resolve target DLLs. In particular, Microsoft Windows supports *Side-by-Side Assembly* [133] and

maintains *Known DLLs* to determine the target DLL fullpath directories without performing iterative directory searching.

Side-by-side assembly. This technique has been provided to mitigate DLL Hell [140]. Using this technique, Windows stores multiple versions of a DLL in the WinSxS directory and loads the desired DLL on demand. For example, when Microsoft Word 2007 loads the Microsoft C runtime library by using its file name (*i.e.*, `MSVCR80.dll`), its full path is determined by a sub-directory of the Windows SxS directory (*i.e.*, `C:\WINDOWS\WinSxS\...\MSVCR80.dll`) without iteratively searching a list of directories. In general, the full path is determined by the existence of its corresponding `Manifest`, an XML document which is usually embedded in the executable. More details can be found in an MSDN article [133].

Known DLLs. The Microsoft Windows operating systems maintain a set of *known DLLs* that correspond to core system DLLs and their load-time dependent ones. The set of core DLLs is determined by the registry key `HKLM\System\CurrentControlSet\Control\Session Manager\KnownDLLs`. If the target DLL is among the known DLLs, its full path is resolved by the directory specified in the `DllDirectory` value located in the registry. In particular, the directory on 32-bit Microsoft Windows family is `%SystemRoot%\system32` by default.

Directory search order. As we mentioned in Section 2.1.1, dynamic component resolution based on filename requires a directory search order, which is determined by system and program settings at runtime. According to MSDN [45], the `LOAD_WITH_ALTERED_SEARCH_PATH` flag, the `SafeDllSearchMode` registry key, and the `SetDllDirectory` system call determine five possible types of directory search orders at runtime, which are *standard search order (SafeDllSearchMode)*, *alternate search order (SafeDllSearchMode)* and *SetDllDire-ctory-based SearchOrder*. Table 3.2 shows the search orders when SafeDllSearchMode is enabled.

Standard search order. The standard search order is the default directory search order in Microsoft Windows, which has two types determined by whether SafeDllSearchMode is enabled. The SafeDllSearchMode was introduced by Microsoft Windows 2000 SP4, and it has been enabled by default since Microsoft Windows XP SP2. For the Standard Search Order of the SafeDllSearchMode, there exist six types of directories to search for DLL resolution (see Table 3.2). If the SafeDllSearchMode is disabled, the priority of the current directory is elevated to the second one.

Search Type	Order
Standard	1. The directory of the application loaded 2. The system directory 3. The 16-bit system directory 4. The Windows directory 5. The current directory 6. The `PATH` environment variable
Alternate	1. The directory specified by *lpFileName* 2. The system directory 3. The 16-bit system directory 4. The Windows directory 5. The current directory 6. The `PATH` environment variable
SetDllDirectory-based	1. The directory of the application loaded 2. The directory specified by *lpPathName* 3. The system directory 4. The 16-bit system directory 5. The Windows directory 6. The `PATH` environment variable

Table 3.2: DLL search types and their directory orders.

Alternate search order. The standard directory search order can be modified when software invokes `LoadLibraryEx` [99] function with the flag `LOAD_WITH_ALTERED_SEARCH_PATH` (*cf.*, Table 3.2). Similar to the standard search order, it is possible to apply the SafeDLLSearchMode to the alternate search order. However, it places the directory of the loading DLL to the first directory to search; the target DLL is specified by its full path, which corresponds to an *lpFileName* parameter of the LoadLibraryEx function.

SetDllDirectory-based search order. Microsoft has provided the `SetDllDirectory` system call to enable developers to manipulate the search order since Microsoft Windows XP SP1. The SetDllDirectory function makes it possible to replace the current directory with an arbitrary directory specified by an *lpPathName* parameter. Also, the current directory can be removed from the search order by invoking the system call with the empty string as the parameter. Note that this search order is independent from the SafeDllSearchMode; the search order is determined as shown in Table 3.2 regardless of the SafeDllSearchMode.

Chained DLL loading. According to Microsoft [4], there exist two types of load-time dependencies among DLLs: implicit dependency and forwarded dependency.

Name	Description
LdrLoadDll	Load a DLL
LdrpApplyFileNameRedirection	Apply the redirection of the DLL specification
LdrpLoadImportModule	Load a chained DLL of the target DLL
LdrpFindLoadedDllByName	Check whether or not a target DLL is loaded
LdrpFindKnownDll	Check whether or not a target DLL is known
LdrpSearchPath	Resolve the fullpath of the target DLL specification

Table 3.3: Instrumented system calls in Windows 7.

Implicit dependency. If a DLL A and a DLL B are linked at compile/link time, and the source code of DLL A calls one or more functions exported from DLL B, DLL A has *implicit dependency* on DLL B. Note that implicit-dependent DLLs are determined by function calls invoked by the source code of the loading DLL. Even though the function is not invoked at runtime, the DLL exporting the function is also loaded. The loading DLL's `Import Directory Table`, one entity of the PE executable file format [106], contains its implicit-dependent DLLs.

Forwarded dependency. While this dependency is similar to implicit dependency, it differs in what the DLL that implements the invoked functions is. For the load-time dependency, the functions that a loading DLL invokes are directly implemented in its dependent DLLs. However, for forwarded dependency, the implementation of the invoked function call simply forwards control to the actual code implemented in another DLL. In this case, the loading DLL has *forwarded dependency* on the DLL containing the forwarded implementation. For example, the `GetLastError` function of `Kernel32.DLL` is forwarded to the `RtlGetLastWin32Error` function of `ntdll.dll`.

Implementation Details

In order to generate profiles for DLL loading behaviors, we utilize *Pin* [100], an open source dynamic binary instrumentation tool. We record a sequence of information on the system calls of interest, image loading, and process/thread identifiers using the functions provided by the tool.

As we mentioned in Section 3.2, the system calls to instrument are determined to cover all possible control-flow paths in the DLL loading procedure. Because this information is not well-documented, we reverse-engineered the `LoadLibraryExW` function of `KernelBase.dll` and the `LdrLoadDll` function of `ntdll.dll` using the IDA Pro Disassembler [69] based on detailed analysis of DLL loadings for Windows 2000 [158]. Table 3.3 describes the system calls instrumented in

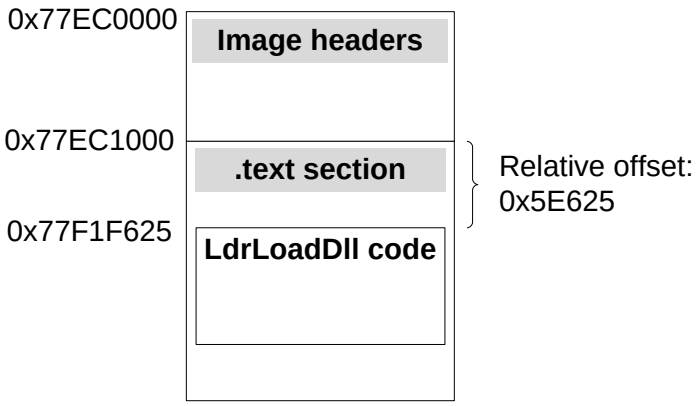


Figure 3.2: Memory layout of `ntdll.dll`.

our implementation. We chose the name of each system call based on the analysis of the disassembler, which uses the Windows symbol package.

Instrumenting a system call requires information about where it is located in the address space such as its starting virtual address. However, this information is difficult to obtain reliably because DLLs follow the PE format and can be relocated in the address space. Also, Address Space Layout Randomization [159] is one of the default configurations of Windows 7, which can randomize the base addresses of the loading images to mitigate memory corruption attacks. To address this problem, we identify the target virtual address by matching its relative offset from the base address of the `.text` section of `ntdll.dll`. Figure 3.2 shows an example of the runtime memory layout of `ntdll.dll`, and the `LdrLoaddll` is located at 0x77F1F625. In this case, the relative offset of the `LdrLoadDll` is 0x5E625 (*i.e.*, 0x77F1F625-0x77EC1000). In the matching, we only focus on the instructions of `ntdll.dll` because all these system calls reside in the file. Because the base address of the `.text` section can be obtained at runtime, we were able to reliably instrument these system calls.

The return value of particular system calls determines the control flow in the loading procedure. For example, if the `LdrpFindLoadedDllByName` returns zero, the specified DLL file has not been loaded yet. To obtain the return value, we also capture the execution of `return` instructions of the system calls in Table 3.3, and retrieve the value of the `eax` register.

To implement our offline profile analysis, we wrote a Python script to extract each DLL loading from the profile and detect unsafe DLL loadings.

```
 1 (b5c,1114) LdrLoadDll dwmapi.dll
 2               C:\Program Files\Mozilla Firefox;
 3               C:\Windows\system32;C:\Windows\system;
 4               C:\Windows;.;$PATH
 5 (b5c,1114) LdrpApplyFileNameRedirection dwmapi.dll
 6               NOT_REDIRECTED
 7 (b5c,1114) LdrpFindLoadedDllByName dwmapi.dll NOT_LOADED
 8 (b5c,1114) LdrpFindKnownDll dwmapi.dll UNKNOWN
 9 (b5c,1114) LdrpSearchPath dwmapi.dll RESOLVED
10 (b5c,1114) IMG_LOAD C:\Windows\system32\dwmapi.dll
11 (b5c,1114) LdrpLoadImportModule msvcrt.dll
12 (b5c,1114) LdrpApplyFileNameRedirection msvcrt.dll
13               NOT_REDIRECTED
14 (b5c,1114) LdrpFindLoadedDllByName msvcrt.dll LOADED
15 (b5c,1114) ...
16 (b5c,1114) LdrpLoadImportModule GDI32.dll
17 (b5c,1114) LdrpApplyFileNameRedirection GDI32.DLL
18               NOT_REDIRECTED
19 (b5c,1114) LdrpFindLoadedDllByName GDI32.dll LOADED
```

Figure 3.3: An unsafe resolution of Firefox 3.6.11

DLL-loading Behavior Profile Example

Figure 3.3 shows part of a generated profile to describe runtime loading procedure of `dwmapi.dll` in Firefox 3.6.11. The profile is composed of two parts. Lines 1–10 represent runtime loading of `dwmapi.dll`, and lines 11–19 correspond to loadings of its load-time dependent DLLs such as `msvcrt.dll` and `GDI32.dll`. The profile provides detailed information on DLL loading. The first and second items in each line describe the process/thread identifier and the loading behavior represented by the corresponding system call name or a tag for image loading, `IMG_LOAD`, respectively. According to the type of the loading behavior, each line contains different information required for the analysis: 1) lines 1–4 contain the target DLL specification given by its file name and the directory search order to be applied for the current DLL loading; 2) lines 5–8 show information on the DLL redirection and the checks for the known DLL and the loaded DLL by the return value of the corresponding system calls, respectively; 3) line 9 shows result of the DLL name resolution; 4) line 10 shows the resolved DLL path for the given DLL specification; and 5) lines 11 and 19 give the scope of the behaviors performed for the chained loading due to the loaded target DLL. Based on this information stored in the profiles, we perform offline analysis to detect unsafe DLL loadings.

```
1 (dd0,98c) LdrLoadDll IMESHARE.DLL
2              C:\Program Files\Microsoft Office\Office14;
3              C:\Windows\system32;C:\Windows\system;
4              C:\Windows;.;$PATH
5 (dd0,98c) LdrpApplyFileNameRedirection IMESHARE.DLL
6              NOT_REDIRECTED
7 (dd0,98c) LdrpFindLoadedDllByName IMESHARE.DLL NOT_LOADED
8 (dd0,98c) LdrpFindKnownDll IMESHARE.DLL UNKNOWN
9 (dd0,98c) LdrpSearchPath IMESHARE.DLL FAILED
```

Figure 3.4: A resolution failure in Microsoft Word 2010.

Example Unsafe DLL Loading

We describe how our technique works by showing examples for each type of unsafe DLL loadings.

Resolution failure. Figure 3.4 shows a resolution failure type in Microsoft Word 2010. The application tries to resolve a DLL specified by `IMESHARE.DLL`. However, the resolution fails because there does not exist a file in the directories determined by the applied directory search order. `LdrpSearchPath` on line 9 is unable to locate the DLL and returns value corresponding to resolution failure.

Unsafe resolution. Figure 3.3 shows an example of unsafe resolution. The DLL specified by `dwmapi.dll` is resolved to `C:\Windows\System32\dwmapi.dll` by checking the file of the specified name located in the directories based on the directory search order. Because the system directory is the second directory to be searched by the OS, placing an arbitrary file of the specified name in the first searched directory (*i.e.*, `C:\Program Files\Mozilla Firefox`) can lead to the hijacking of the intended DLL loading.

3.3.2 Linux Distribution

This section presents necessary background on loading of Shared Object (SO) on Linux and the implementation details of our tool for Ubuntu 10.04. Although we implemented our tool for Ubuntu, our implementation technique can also be directly applied to other Linux distributions such as Fedora as dynamic loading is governed by the Linux kernel.

Order	Description
1. DT_RPATH	`.dynamic` section attribute
2. LD_LIBRARY_PATH	Environment variable
3. DT_RUNPATH	`.dynamic` section attribute
4. CACHED_DIR	Directories cached by `/etc/ld.so.conf`
5. PATH	Environment variable

Table 3.4: Directory search order in Linux.

Background on SO Loading

The library `ld-linux.so.2` [96] serves as a dynamic loader on Linux. Whenever a dynamic loading is requested, the loader resolves the target file based on the specification and loads the file. The loader resolves the target file based on the approach discussed in Section 2.1.1. If the fullpath is specified, it simply determines the target file. If the filename is specified, the OS iterates through the search directories to locate the file with the specified name. For more details on this resolution process, please refer to the `_dl_map_object` function in `dl-load.c`, which is part of the GNU C library [54].

Directory search order. During dynamic loading, the OS determines the directory search order based on the current *system configuration* and *particular attributes of the running executable file* at runtime. Table 3.4 shows an example search order.

Environment variables. Linux has a number of environment variables (such as `LD_LIBRARY_PATH` and `PATH`) that store a sequence of the directories searched by the loader. Thus, an application can manipulate these variables to specify arbitrary directory search orders.

Cached SO directories. As Windows maintains the set of known DLLs, Linux caches a list of the directories where known shared objects are located in the `/etc/ld.so.conf` file. When the filename of a known object is given, the OS resolves its fullpath by using this information.

The *`.dynamic` section attributes.* The Executable and Linkable Format (ELF) [49] has been used as the executable file format in Linux. An ELF file such as an SO file consists of a set of *sections*. Each section contains important data for program execution. For example, the `.text` section stores program code.

The ELF format has the `.dynamic` section, which provides the OS with dynamic loading information. In particular, this section has a set of attributes related to dynamic loading. Figure 3.5

```
$ readelf -d /usr/bin/amarok
Dynamic section at offset 0xfed8 contains 30 entries:
  Tag        Type      Name/Value
 0x00000001 (NEEDED)  Shared library: [libkdeui.so.5]
 0x00000001 (NEEDED)  Shared library: [libamaroklib.so.1]
 0x00000001 (NEEDED)  Shared library: [libQtGui.so.4]
 0x00000001 (NEEDED)  Shared library: [libkdecore.so.5]
 0x00000001 (NEEDED)  Shared library: [libQtCore.so.4]
 0x00000001 (NEEDED)  Shared library: [libstdc++.so.6]
 0x00000001 (NEEDED)  Shared library: [libgcc_s.so.1]
 0x00000001 (NEEDED)  Shared library: [libc.so.6]
 0x0000000f (RPATH)   Library rpath: [/usr/lib]
 0x0000001d (RUNPATH) Library runpath: [/usr/lib]
 ...
```

Figure 3.5: The .dynamic section of Amarok 2.3.0.

shows a snippet of the `.dynamic` section of Amarok 2.3.0 (a free software music player).

To support user-defined search directories, the ELF format has two attributes in the `.dynamic` section, `DT_RPATH` and `DT_RUNPATH`, which specify sequences of the directories searched by the executable at runtime. In Figure 3.5, the `/usr/lib` directory is specified for both attributes tagged with `RPATH` and `RUNPATH`.

Chained SO loading. As we mentioned in Section 2.1.1, when a shared object is loaded, its dependent shared objects are also loaded. In Linux, the specifications of these dependent objects are stored in the `DT_NEEDED` attributes of the `.dynamic` section of the loaded object. For example, `/usr/bin/amarok` has eight dependent objects tagged with `NEEDED` in Figure 3.5. Thus, the loader recursively resolves the dependent objects based on the specifications for chained loading. More details can be found in the ELF manual [49].

Implementation Details

To implement the tool for Linux, we adopt the same method as our Windows implementation. We utilized *Pin* [100] to capture the system-level behavior for loading shared objects and implemented the detection of unsafe loadings as Python scripts.

To determine which system calls to instrument, we reverse-engineered `ld-linux.so.2` based on the source code of the GNU C library [54]. Table 3.5 shows the system calls we instrumented to generate the profiles. Besides the entry/exit points of these system calls, we also instrument call sites to these system calls. For example, the `_dl_map_object` has several call sites to `open_path`, and

Name	File	Description
dl_open_worker	dl-open.c	Perform dynamic loading
_dl_map_object	dl-load.c	Perform dynamic loading (internal function)
open_path	dl-load.c	Iterate through the search directories to locate the specified file
openaux	dl-deps.c	Perform chained loading

Table 3.5: Instrumented system calls in Linux

each of them corresponds to a directory search order in Table 3.4. To instrument a virtual address of interest, we adopt a similar approach as in our Windows implementation—we match the relative offset of the address from the `.text` section of `ld-linux.so.2` at runtime.

Unlike Windows, Linux does not invoke a system call to check whether or not the specified file is loaded. Instead, the `_dl_map_object` iterates over a list of the loaded objects and returns information of the matched file if the specified file exists on the search list. To record the result of this search iteration, we instrumented the instruction to return the information: if the instruction is captured at runtime, the specified SO is already loaded. We store such information in the profile by using a keyword `check_loaded` (*cf.*, Figure 3.6).

SO-loading Behavior Profile Example

Figure 3.6 shows a snippet of the generated behavior profile for loading `libnss_compact.so.2` in Evolution 2.28.3. This profile consists of two parts, lines 1–5 and lines 6–14. The first part performs dynamic loading of `libnss_compact.so.2`. To load this file, the OS first checks whether it has already been loaded (line 2). Because the file has not been loaded, the OS iterates through a sequence of the directories determined by Table 3.4. Note that only `DT_RPATH` and `CACHED_DIR` affect the runtime construction of the directory sequence. As the file is cached, the OS resolves it based on the cached directory, `/lib/tls/i686/cmov/`. In the second part, the OS performs chained shared object loading. In particular, `libnss_compact.so.2` has three dependent objects: `libnsl.so.1`, `libc.so.6`, and `ld-linux.so.2`. During this chained loading, only the first dependent SO is loaded, because the other two are already loaded.

```
 1 (9ce, 9ce) dlopen libnss_compat.so.2
 2 (9ce, 9ce) check_loaded NOT_LOADED
 3 (9ce, 9ce) open_path DT_RPATH /usr/lib/evolution/2.28/:
 4 (9ce, 9ce) open_path CACHED_DIR CACHED
 5 (9ce, 9ce) IMAGE_LOAD /lib/tls/i686/cmov/libnss_compat.so.2
 6 (9ce, 9ce) openaux libnsl.so.1
 7 (9ce, 9ce) check_loaded NOT_LOADED
 8 (9ce, 9ce) open_path DT_RPATH /usr/lib/evolution/2.28/:
 9 (9ce, 9ce) open_path CACHED_DIR CACHED
10 (9ce, 9ce) IMAGE_LOAD /lib/tls/i686/cmov/libnsl.so.1
11 (9ce, 9ce) openaux libc.so.6
12 (9ce, 9ce) check_loaded LOADED
13 (9ce, 9ce) openaux ld-linux.so.2
14 (9ce, 9ce) check_loaded LOADED
```

Figure 3.6: An unsafe resolution of Evolution 2.28.3.

```
1 (cac, cac) dl_open_worker libXfixes.so.1
2 (cac, cac) open_path CACHED_DIR NOT_CACHED
3 (cac, cac) open_path PATH /lib/:/usr/lib/:
             /lib/i486-linux-gnu/:/lib/i486-linux-gnu/:
             FAILED
```

Figure 3.7: A resolution failure of Konqueror 4.4.2.

Example Unsafe SO Loadings

Resolution failure. Figure 3.7 shows a failure to load `libXfixes.so.1` in Konqueror 4.4.2. In particular, the OS locates the non-cached file in the directories specified by the `PATH` variable. In this case, if any file named `libXfixes.so.1` exists in one of the directories, this loading can be hijacked.

Unsafe resolution. Figure 3.6 is an example of unsafe resolution. As we discussed in Section 3.3.2, the directory `/usr/lib/evolution/2.28/`, specified by `DT_RPATH`, is checked by the OS before resolving the fullpath of the specified file `libnss_compact.so.2`. Thus, this loading can be hijacked by placing an arbitrary file with the specified name in that directory.

3.4 Evaluation

In this section, we evaluate unsafe component loadings on Microsoft Windows and Linux. For each platform, we detect unsafe component loadings in a diverse selection of popular applications. We structure our analysis of the detection results to answer the following research questions:

RQ1: How prevalent and severe are unsafe component loadings on Microsoft Windows? (*Sec-*

Software	Windows XP					Windows Vista					Windows 7				
	Failed			Unsafe		Failed			Unsafe		Failed			Unsafe	
	Fullpath	Filename		Filename		Fullpath	Filename		Filename		Fullpath	Filename		Filename	
	T	T	C	T	C	T	T	C	T	C	T	T	C	T	C
MS Office															
Excel 2010	0	1	0	10	14	0	1	0	6	8	0	2	0	14	9
OneNote 2010	0	0	0	7	12	0	1	0	26	21	0	0	0	34	8
Outlook 2010	2	2	0	27	24	2	2	0	22	20	2	1	0	15	23
PowerPoint 2010	1	2	0	16	20	1	2	0	16	16	1	2	0	19	13
Publisher 2010	2	1	0	17	20	2	2	0	11	23	2	1	0	18	16
Word 2010	2	2	0	10	17	1	1	0	16	11	15	2	0	18	10
Sub total	7	8	0	87	107	6	9	0	97	99	20	8	0	118	79
Web Browser															
Chrome 6.0.472.63	1	1	0	22	17	1	1	0	9	8	1	1	0	30	13
Firefox 3.6.10	1	1	0	18	5	1	1	0	19	14	1	0	1	16	10
IE 8.0 / 9.0 Beta	0	0	0	20	17	0	0	0	17	15	0	0	0	21	6
Opera 10.63	2	1	0	9	8	0	1	0	6	22	0	1	0	14	9
Safari 5	0	1	0	13	59	0	0	0	9	36	0	0	0	19	30
Sub total	4	4	0	82	106	2	3	0	60	95	2	2	1	100	68
PDF Reader															
Acrobat Reader 9.4.0	0	0	0	6	9	0	0	0	17	11	0	0	0	5	5
Foxit Reader 4.2	0	0	0	14	10	0	1	0	20	12	0	0	0	9	17
Sub total	0	0	0	20	19	0	1	0	37	23	0	0	0	14	22
Messenger															
Google Talk Beta	0	1	0	21	10	0	0	0	14	8	0	1	0	28	19
Pidgin 2.7.3	0	0	2	11	30	0	0	2	10	33	0	0	2	13	35
Skype 4.2.0	0	0	0	35	20	0	0	0	26	23	0	0	0	34	11
Windows Live 2011	0	2	0	28	14	0	3	0	33	22	0	2	0	43	50
Yahoo! 10.0	0	0	0	24	20	0	2	0	38	28	0	1	0	45	35
Sub total	0	3	2	119	94	0	5	2	121	114	0	4	2	163	150
Image Viewer															
Irfanview 4.27	0	0	0	8	10	0	0	0	16	6	0	0	0	2	2
Picasa 3.8	0	0	0	20	12	1	1	0	15	13	1	0	0	18	26
Sub total	0	0	0	28	22	1	1	0	31	19	1	0	0	20	28
Multimedia Player															
iTunes 10.0.1	0	1	0	38	76	0	1	0	27	56	0	1	0	38	46
Media Player 12	0	2	0	27	19	0	1	0	20	14	0	1	0	31	7
Quicktime 7.6.8	0	0	0	26	33	0	1	0	20	41	0	0	0	27	30
RealPlayer SP 1.1.5	1	0	0	17	18	2	7	0	30	32	2	3	0	26	20
Winamp 5.58	4	2	0	16	9	1	0	0	21	20	4	1	0	12	9
Sub total	5	5	0	124	155	3	10	0	118	163	6	6	0	134	112
Others															
Google Desktop 5.9	0	0	0	10	5	0	0	0	8	1	0	0	0	10	5
Google Earth 5.2.1	1	4	0	24	27	1	4	0	15	31	1	4	0	23	14
Sub total	1	4	0	34	32	1	4	0	23	32	1	4	0	33	19
Total	17	24	2	494	535	13	33	2	487	545	30	24	3	582	478

Table 3.6: Number of detected unsafe DLL loadings.

tion 3.4.1)

RQ2: How prevalent and severe are unsafe component loadings on Linux? (*Section 3.4.2*)

RQ3: What are the implications of our findings? (*Section 3.4.3*)

RQ4: How does our detection technique compare to related work? (*Section 3.4.4*)

Software	Windows XP			Windows Vista			Windows 7		
	Target		Chained	Target		Chained	Target		Chained
	Fullpath	Filename	Filename	Fullpath	Filename	Filename	Fullpath	Filename	Filename
MS Office									
Excel 2010	0 / 31	11 / 13	14 / 28	0 / 22	7 / 7	8 / 14	0 / 35	16 / 17	9 / 23
OneNote 2010	0 / 20	7 / 7	12 / 17	0 / 28	27 / 27	21 / 27	0 / 36	34 / 36	8 / 12
Outlook 2010	2 / 81	29 / 30	24 / 42	2 / 63	24 / 24	20 / 31	2 / 68	16 / 19	23 / 30
PowerPoint 2010	1 / 52	18 / 24	20 / 41	1 / 43	18 / 22	16 / 31	1 / 46	21 / 27	13 / 29
Publisher 2010	2 / 49	18 / 21	20 / 41	2 / 38	13 / 15	23 / 32	2 / 54	19 / 27	16 / 30
Word 2010	2 / 59	12 / 18	17 / 36	1 / 37	17 / 20	11 / 19	15 / 85	20 / 22	10 / 18
Sub total	7 / 292	95 / 113	107 / 205	6 / 231	106 / 115	99 / 154	20 / 324	126 / 148	79 / 142
Web Browser									
Chrome 6.0.472.63	1 / 20	23 / 23	17 / 22	1 / 19	10 / 10	8 / 12	1 / 24	31 / 34	13 / 18
Firefox 3.6.10	1 / 16	19 / 20	5 / 27	1 / 24	20 / 20	14 / 39	1 / 32	16 / 17	10 / 30
IE 8.0 / 9.0 Beta	0 / 25	20 / 21	17 / 27	0 / 18	17 / 20	15 / 15	0 / 26	21 / 24	6 / 13
Opera 10.63	2 / 20	10 / 11	8 / 9	0 / 22	7 / 8	22 / 29	0 / 27	15 / 17	9 / 12
Safari 5	0 / 11	14 / 16	59 / 67	0 / 17	9 / 11	36 / 45	0 / 20	19 / 21	30 / 41
Sub total	4 / 92	86 / 91	106 / 152	2 / 100	63 / 69	95 / 140	2 / 129	102 / 113	68 / 114
PDF Reader									
Acrobat Reader 9.4.0	0 / 17	6 / 9	9 / 15	0 / 22	17 / 26	11 / 29	0 / 14	5 / 10	5 / 12
Foxit Reader 4.2	0 / 12	14 / 14	10 / 26	0 / 24	21 / 21	12 / 30	0 / 14	9 / 9	17 / 19
Sub total	0 / 29	20 / 23	19 / 41	0 / 46	38 / 47	23 / 59	0 / 28	14 / 19	22 / 31
Messenger									
Google Talk Beta	0 / 19	22 / 23	10 / 27	0 / 12	14 / 14	8 / 25	0 / 25	29 / 31	19 / 21
Pidgin 2.7.3	0 / 71	11 / 13	32 / 52	0 / 71	10 / 12	35 / 55	0 / 78	13 / 16	37 / 58
Skype 4.2.0	0 / 17	35 / 37	20 / 23	0 / 18	26 / 26	23 / 34	0 / 36	34 / 35	11 / 26
Windows Live 2011	0 / 33	30 / 40	14 / 63	0 / 44	36 / 39	22 / 70	0 / 54	45 / 48	50 / 70
Yahoo! 10.0	0 / 31	24 / 28	20 / 62	0 / 35	40 / 50	28 / 70	0 / 48	46 / 55	35 / 57
Sub total	0 / 171	122 / 141	96 / 227	0 / 180	126 / 141	116 / 254	0 / 241	167 / 185	152 / 232
Image Viewer									
Irfanview 4.27	0 / 21	8 / 8	10 / 26	0 / 28	16 / 16	6 / 20	0 / 9	2 / 2	2 / 4
Picasa 3.8	0 / 20	20 / 22	12 / 41	1 / 21	16 / 17	13 / 52	1 / 40	18 / 18	26 / 38
Sub total	0 / 41	28 / 30	22 / 67	1 / 49	32 / 33	19 / 72	1 / 49	20 / 20	28 / 42
Multimedia Player									
iTunes 10.0.1	0 / 46	39 / 41	76 / 102	0 / 45	28 / 29	56 / 70	0 / 49	39 / 42	46 / 57
Media Player 12	0 / 27	29 / 38	19 / 25	0 / 14	21 / 21	14 / 17	0 / 29	32 / 34	7 / 16
Quicktime 7.6.8	0 / 36	26 / 36	33 / 45	0 / 44	21 / 25	41 / 57	0 / 46	27 / 35	30 / 40
RealPlayer SP 1.1.5	1 / 31	17 / 17	18 / 26	2 / 141	37 / 43	32 / 39	2 / 59	29 / 31	20 / 26
Winamp 5.58	4 / 139	18 / 27	9 / 18	1 / 73	21 / 21	20 / 28	4 / 83	13 / 14	9 / 19
Sub total	5 / 279	129 / 159	155 / 216	3 / 317	128 / 139	163 / 211	6 / 266	140 / 156	112 / 158
Others									
Google Desktop 5.9	0 / 11	10 / 13	5 / 12	0 / 7	8 / 11	1 / 6	0 / 13	10 / 17	5 / 9
Google Earth 5.2.1	1 / 20	28 / 32	27 / 38	1 / 21	19 / 22	31 / 42	1 / 21	27 / 30	14 / 22
Sub total	1 / 31	38 / 45	32 / 50	1 / 28	27 / 33	32 / 48	1 / 34	37 / 47	19 / 31
Total	17 / 935	518 / 602	537 / 958	13 / 951	520 / 577	547 / 938	30 /1071	606 / 688	480 / 750

Table 3.7: Ratio of unsafe to total DLL loadings.

3.4.1 Evaluation Results on Windows

We evaluate the prevalence and severity of unsafe DLL loadings in 27 popular applications on Windows XP SP3, Vista SP2, and Windows 7. The conference version [94] of this chapter reports our evaluation results on older versions of the test subjects on Windows XP SP3 and Vista SP1.

Analysis of Unsafe DLL Loadings

In our evaluation, the detection of unsafe DLL loadings is performed with the administrator privilege because 1) we aim to detect all possible unsafe DLL loadings to evaluate the worst case, and 2) most Windows users have the administrator privilege [141], in contrast to Unix/Linux-based operating systems.

To collect the runtime traces, we installed the applications and necessary drivers (*e.g.*, printer) on the default configurations of the operating systems. Afterwards, we executed them one by one with relevant inputs (*e.g.*, `.docx` file for Microsoft Word 2010) and collected a single trace per application. For example, we extracted the runtime traces of the web browsers by accessing `http://www.google.com`.

Table 3.6 shows the number of unsafe DLL loadings detected from a few different types of major applications on Microsoft Windows family. In particular, we classify detected failed and unsafe resolutions in terms of the specification type (*i.e.*, fullpath or filename) and the phase at which the unsafe loadings happen. The columns labeled `T` and `C` correspond to target and chained component loadings, respectively. Note that the `C` column is missing for fullpath. This is because components for the chained loading are specified by their filenames. According to the table, unsafe DLL loadings are common programming mistakes in developing these applications. We found more than 3,200 instances of unsafe dynamic loadings: 1,072 under XP, 1,080 under Vista, and 1,117 under Windows 7. Considering the types of these unsafe DLL loadings, unsafe resolution is responsible for almost all of them. In particular, unsafe resolution in Windows XP, Vista, and Windows 7 corresponds to 95.9% (1,029/1,072), 95.5% (1,032/1,080), and 94.9% (1,060/1,117) of the total unsafe loadings, respectively. Next we give a detailed analysis of each type of the unsafe DLL loadings.

Table 3.7 illustrates the ratio of unsafe to the total number of DLL loadings captured during our evaluation. Similar to Table 3.6, we classify each DLL loading in terms of the specification type and the phase and specify corresponding ratio in the table. One interesting finding is that filename-based target specifications are generally unsafe. For example, 90% (520/577) of the filename-based target specifications in Windows Vista lead to unsafe component loadings. Thus, to mitigate this issue, it is necessary to exert care in specifying filenames for dynamic loading. Section 3.5 discusses possible

DLL Type	FULLPATH			FILENAME		
	XP	Vista	7	XP	Vista	7
Application	6	7	7	6	7	9
Third-party component	7	4	7	13	13	12
Language support	2	2	16	0	0	0
Unsupported	2	0	0	7	15	6

Table 3.8: Types of target DLLs whose resolutions fail.

mitigation techniques in detail.

Resolution failures. Table 3.8 shows types of target DLLs whose resolutions fail. In particular, for the fullpath and filename specifications, there exist four types of target DLLs: application DLL, third-party component DLL, language support DLL, and unsupported system DLL.

Application DLL. Many applications do not include application-specific DLLs in their releases, which can cause resolution failures of these libraries. For example, `Google Earth 5.2.1` tries to load `collada.dll` in the application directory when it starts up, but such a library is not included in the release.

Third-party component DLL. Third-party components embedded in applications can also cause DLL resolution failures. There are two main reasons for this: 1) difference in the directory search order between the application and the component, and 2) the loadings of missing DLLs by the components.

For the first reason, when an application loads a third-party component, the applied directory search order for the resolution is determined by the setting of the running application. Because the intended directory search order for the component can be different from the applied one, the DLL resolution by the component can fail. For example, `Google Desktop` registers a `Google Desktop Office Addin` to Microsoft Word and PowerPoint, and it is loaded when these applications run. During the loading procedure, the component tries to load a DLL file `GoogleDesktopCommon.dll`, which is located in the directory of `Google Desktop`. However, because the applied directory search order does not contain this directory, this resolution fails.

Similar to resolution failures of an application, third-party components may attempt to load DLLs that do not exist on the system due to careless programming. For example, Microsoft PowerPoint 2010 tries to load driver files of the printers installed on the system during their startup.

DLL-Hijacking Directory	XP		Vista		Windows 7	
	T	C	T	C	T	C
Application	494	488	487	477	582	430
Application library	81	122	63	103	94	82
System	1	4	1	3	1	14
Part of $PATH	1	2	1	1	1	10
Plug-in	0	6	0	7	0	9
WBEM	0	11	0	11	0	6
Driver	0	13	0	11	6	6
System-hook source	0	11	0	20	0	13

Table 3.9: Types of DLL-hijacking directories.

However, some HP printer drivers try to load the non-existing `HPProfiler.dll` during the driver loading process, which causes the resolution failure.

Language-support DLL. Many applications load resource files for language-support, but these files may not exist on the system. For example, when Microsoft Outlook 2010 runs on the Korean version of Microsoft Windows XP Professional SP3, it loads `SOCIALCONNECTORKOR.dll` in its application directory. However, the release of Microsoft Outlook does not contain such a file.

Unsupported system DLL. Newer versions of Windows provide some DLL files to support new features. Because these DLLs do not exist on older versions of Windows, it is necessary to consider the version of the current operating system when loading these files. However, many applications developed for the Windows platforms usually do not consider this issue. The examples of these cases are as follows:

- Many applications for Windows Vista try to load `peerdist.dll`, which is a DLL for BranchCache Client Library in Windows 7.
- `Winamp 5.58` loads `DWMAPI.dll`, a DLL for Windows Manager API in Windows Vista, even if the current operating system is Windows XP.

Unsafe resolution. Tables 3.9 and 3.10 show distributions of types of DLL-hijacking and resolved directories. These results indicate that most unsafe resolutions of system DLLs can be hijacked from the directories of the applications loading them.

Table 3.9 also shows that there exist types of DLL-hijacking directories that are not related to the application such as the plug-in directory. This is because the target DLL is specified by its full

Resolved Directory	XP	Vista	7
System directory	953	956	975
Application library directory	76	76	85

Table 3.10: Types of resolved directories.

path, and the alternate search order in Table 3.2 is applied to load its load-time-dependent DLLs, which searches the directory of the target DLL first. For example, Yahoo! Messenger 10.0 loads `C:\Windows\System32\Macromed\Flash\Flash10k.ocx` to use Flash. After loading the `Flash10k.ocx`, its load-time dependent DLLs such as `DSOUND.dll` are resolved to the file in the system directory. In this case, the Flash directory can serve as a DLL-hijacking directory, because the directory is searched before the resolution of the target DLL based on the applied search order.

Severity

In this section, we evaluate exploitability of unsafe component loadings in terms of local and remote attacks. Local attacks assume that attackers can access the local file system on a victim host, while remote attacks assume that attackers can only send data to the victim user.

Local attacks. As we mentioned in Section 2.1.1, unsafe DLL loading can be performed by placing a file with the specified name in the DLL-hijacking directories. To exploit this security vulnerability for local attacks, attackers require write permission to the DLL-hijacking directory. According to Tables 3.8 and 3.9, most of the directories are not writable by non-admin users. Therefore, if attackers do not have administrator privilege, most local attacks can be prevented. However, according to Microsoft [141], most Windows users run with administrative privilege. Because of this fact, unsafe DLL loadings should still be considered serious security issues.

Remote attacks. To accomplish remote attacks exploiting unsafe component loadings, attackers need to place malicious files in the DLL-hijacking directories from remote sites. However, accessing the file system of a remote host is generally prohibited. For example, the system directory is not accessible remotely unless the directory is shared to the remote user or the system is exploited by other vulnerabilities to enable this. Because of the difficulty in remote exploitation, unsafe component loadings have not been considered serious security threats. However, as we mentioned in

Section 2.1.3, several remote attack vectors based on unsafe component loading have been recently discovered.

To find remote attacks on Microsoft Windows, we focus on unsafe DLL loadings caused by the following three conditions: *resolution failure*, *filename specification*, and *standard or alternate search order*. According to the directory search orders discussed in Table 3.2, this type of unsafe DLL loading makes the OS check the current directory corresponding to "`.`" during DLL resolution. In this case, the directory may be writable from the remote site because of software bugs. The blended threat combined with the Safari's Carpet Bomb attack discussed in Section 2.1.3 exploits this flaw. In particular, when Internet Explorer 7 tries to resolve `sqmapi.dll`, the current directory is checked before the resolution and corresponds to the Desktop directory. This makes the program load and execute malicious DLL files on the Desktop directory, which are downloaded through the Carpet Bomb attack.

Based on this observation, we detect potential remote attacks by checking whether or not the current directory is writable by remote users when a resolution failure based on the filename specification happens. In this evaluation, we consider the following two types as remotely writable directories: *directory sent by remote users* and the *Desktop directory*. For the first directory type, attackers can send arbitrary directory structures by using archive files similar to malware propagation via e-mail. Considering the Desktop directory, we assume that the Carpet Bomb attack is possible. Table 3.11 shows detailed information on the detected attack vectors. In the table, the `o` corresponds to the exploitable case, while the `x` corresponds to the non-exploitable case.

Shortcut with component. The current directory of applications run via their shortcuts may be the same directory as the shortcuts at the point of the resolution failure. In this case, the shortcut directory can serve as the DLL-hijacking directory for remote code execution. For example, RealPlayer SP 1.1.5 run via its shortcut on Windows Vista and 7 has a flaw where it loads `SHDOCLC.DLL` located in the same directory as the shortcut. This vulnerability can lead to remote code execution attacks through social engineering attacks. Furthermore, this type of attack can be combined with the Carpet Bomb attack because the usual location of the shortcut is the desktop directory.

One interesting discovery is the attack caused by third-party component loading. For example, when Opera 10.63 runs via its shortcut on the host where Google Desktop is installed, it loads

Application	OS	Filename	Shortcut	Document	Third-party component
Foxit Reader 4.2	Vista	peerdist.dll	o	o	
MS PowerPoint 2010	XP, Vista, 7	HPProfiler.dll	o	o	HP printer driver
		GoogleDesktopCommon.dll	o	o	Google Desktop
MS Publisher 2010	XP, Vista, 7	HPProfiler.dll	x	o	HP printer driver
	Vista	peerdist.dll	x	o	
MS Word 2010	XP, Vista	GoogleDesktopCommon.dll	o	o	Google Desktop
	7	GoogleDesktopCommon.dll	x	o	Google Desktop
	XP	HPProfiler.dll	o	o	HP printer driver
	7	IMESHARE.DLL	x	o	
Opera 10.63	XP, Vista, 7	GoogleDesktopCommon.dll	o	o	Google Desktop
RealPlayer SP 1.1.5	Vista, 7	SHDOCLC.DLL	o	x	
		rio500.dll	o	x	
		rio300.dll	o	x	
	Vista	peerdist.dll	o	x	
Windows Live Messenger 2011	XP	dwmapi.dll	o	x	
		GoogleDesktopCommon.dll	o	x	Google Desktop

Table 3.11: Remote attacks based on unsafe component loadings.

`GoogleDesktopCommon.dll` on its startup. This vulnerability shows that third-party components can cause software hosting them to perform unsafe component loading, which can be exploited by attackers for remote code execution.

Document with component. Opening arbitrary document files can lead to serious security holes for the remote attacks based on the unsafe component loadings. For example, if a user opens a document for MS Word 2010 in Windows 7, `IMESHARE.DLL` located in the same directory as the document is loaded. This flaw can be exploited in various types of social engineering attacks (*e.g.*, malicious email attachment).

As we mentioned above, third-party components can cause serious security vulnerabilities in software. According to Table 3.11, Microsoft Word and PowerPoint 2010 suffer from security vulnerabilities caused by third-party components, which lead to remote code execution attacks. In particular, loading the `Google Desktop Office Addin` and the `HP printer driver` fails to resolve particular DLLs when the programs open documents, and the current directory at that point is the same directory as the opened documents. This security hole allows attackers to make the software load the DLLs from remote sites when the victim opens the document. According to our analysis, Microsoft Word and PowerPoint 2007 also have the same security holes [94]. We reported this issue to the Microsoft Security Response Center and have been working with Microsoft in

Software	Generation (s)	Analysis (ms)
Excel 2010	51	31
OneNote 2010	36	13
Outlook 2010	116	35
PowerPoint 2010	61	23
Publisher 2010	51	26
Word 2010	83	28

Table 3.12: Execution time for analyzing MS Office 2010.

collaboration with Google and HP to develop security patches.

For Foxit Reader 4.2, the directory containing the opened PDF document can be considered the DLL-hijacking directory due to a resolution failure of `peerdist.dll` in Windows Vista. This flaw allows attackers to perform remote code execution attacks by sending archives of a PDF document and a malicious `peerdist.dll` to remote users.

Performance

To evaluate the performance of our technique, we measure the execution time of each phase for analyzing MS Office products on Windows 7 running on a Core2 Duo 2.40GHz processor with 4GB RAM. Table 3.12 shows the execution time for the profile generation and analysis phases of the analyzed applications. In the evaluation, we use default documents as inputs to the analyzed programs. Our results show that our technique is practical and can be effectively applied for analyzing real-world programs such as MS Office.

3.4.2 Evaluation Results on Linux

This section discusses our evaluation of the prevalence and severity of unsafe loadings of SO files on 24 popular applications on Ubuntu 10.04. To collect the runtime traces, we installed the applications under the default configuration of the OS and adopted the same strategy as our evaluation on Windows (*cf.*, Section 3.4.1).

Software	Failed			Unsafe	
	Fullpath	Filename		Filename	
	T	T	C	T	C
Email Client					
Balsa 2.4.1	0	0	0	0	0
Evolution 2.28.3	1	1	0	4	155
Kmail 1.13.2	0	36	0	2	10
Thunderbird 3.0.8	0	0	0	0	67
Sub total	1	37	0	6	232
Web Browser					
Chrome 6.0.472.63	0	0	0	0	0
Firefox 3.6.10	1	0	0	5	103
Konqueror 4.4.2	0	39	0	4	8
Opera 10.62.6438	0	1	0	0	0
Seamonkey 2.0.8	0	0	0	3	113
Sub total	1	40	0	12	224
PDF Reader					
Acrobat Reader 9.3.4	0	0	0	5	70
Foxit Reader 1.1	0	0	0	0	0
Sub total	0	0	0	5	70
Messenger					
Empathy 2.30.2	0	0	0	0	0
Pidgin 2.6.6	0	1	0	0	1
Skype 2.1	0	36	0	0	0
Sub total	0	37	0	0	1
Multimedia Player					
Amarok 2.3.0	34	38	0	2	9
RealPlayer 11.0.2.1744	1	0	0	0	0
Rhythmbox 0.12.8	0	0	0	0	0
Totem 2.30.2	0	0	0	0	0
Sub total	35	38	0	2	9
Text Editor					
Emacs 24.3.1	0	0	0	0	0
Gvim 7.2	0	0	0	0	0
Sub total	0	0	0	0	0
Others					
Brasero 2.30.2	0	1	0	0	0
Cheese 2.30.1	0	0	0	0	0
Filezilla 3.3.1	0	1	0	0	0
Gimp 2.6	0	0	0	0	0
Sub total	0	2	0	0	0
Total	37	154	0	25	536

Table 3.13: Number of detected unsafe SO loadings.

Software	Target		Chained
	Fullpath	Filename	Filename
Email Client			
Balsa 2.4.1	0 / 0	0 / 0	0 / 1
Evolution 2.28.3	1 / 32	5 / 5	155 / 178
Kmail 1.13.2	0 / 35	38 / 94	10 / 179
Thunderbird 3.0.8	0 / 0	0 / 0	67 / 74
Sub total	1 / 67	43 / 99	232 / 432
Web Browser			
Chrome 6.0.472.63	0 / 15	0 / 5	0 / 96
Firefox 3.6.10	1 / 58	5 / 5	103 / 112
Konqueror 4.4.2	0 / 33	43 / 100	8 / 126
Opera 10.62.6438	0 / 8	1 / 6	0 / 67
Seamonkey 2.0.8	0 / 73	3 / 3	113 / 125
Sub total	1 / 187	52 / 119	224 / 526
PDF Reader			
Acrobat Reader 9.3.4	0 / 14	5 / 5	70 / 84
Foxit Reader 1.1	0 / 3	0 / 2	0 / 70
Sub total	0 / 17	5 / 7	70 / 154
Messenger			
Empathy 2.30.2	0 / 9	0 / 2	0 / 124
Pidgin 2.6.6	0 / 81	1 / 5	1 / 137
Skype 2.1	0 / 14	36 / 97	0 / 96
Sub total	0 / 104	37 / 104	1 / 357
Multimedia Player			
Amarok 2.3.0	34 / 101	40 / 102	9 / 172
RealPlayer 11.0.2.1744	1 / 67	0 / 2	0 / 70
Rhythmbox 0.12.8	0 / 32	0 / 2	0 / 118
Totem 2.30.2	0 / 35	0 / 2	0 / 109
Sub total	35 / 235	40 / 108	9 / 469
Text Editor			
Emacs 24.3.1	0 / 9	0 / 3	0 / 80
Gvim 7.2	0 / 9	0 / 2	0 / 104
Sub total	0 / 18	0 / 5	0 / 184
Others			
Brasero 2.30.2	0 / 35	1 / 4	0 / 90
Cheese 2.30.1	0 / 8	0 / 2	0 / 77
Filezilla 3.3.1	0 / 8	1 / 4	0 / 81
Gimp 2.6	0 / 74	0 / 2	0 / 85
Sub total	0 / 125	2 / 10	0 / 333
Total	37 / 753	179 / 451	536 / 2455

Table 3.14: Ratio of unsafe to total SO loadings.

SO-hijacking Directory	Failed		Unsafe
	Fullpath	Filename	Filename
/lib/*	0	616	0
/opt/*	13	0	138
/usr	0	0	16
/usr/lib/*	7	620	835
/usr/lib64/*	4	0	0
/usr/local/lib/*	7	0	0
/usr/local/lib64/*	4	0	0
∼/*	2	0	0

Table 3.15: Types of SO-hijacking directories.

Order	Configured	Failed	Unsafe
DT_RPATH	T	2	140
LD_LIBRARY_PATH	T	0	366
RUNPATH	T	2	35
CACHED_DIR	F	310	0
PATH	T	308	0

Table 3.16: Insecure configuration for unsafe SO loadings.

Analysis of Unsafe SO Loadings

Tables 3.13 and 3.14 illustrate the prevalence of unsafe SO loadings and the ratio of unsafe to the total number of SO loadings, respectively. The columns labeled `T` and `C` in Table 3.13 correspond to target and chained component loadings respectively. According to the table, unsafe SO loadings are also prevalent in Linux applications. During the evaluation, we detected 752 instances of unsafe SO loadings in 14 out of the 24 test subjects. Also, we analyzed the detected unsafe loadings and show the types of the SO-hijacking directories in Table 3.15. Note that multiple directories can be used for hijacking the loading of an SO file.

As discussed in Section 3.3.2, developers can control the directory search order by adapting current system configuration and particular attributes of the running executable file at runtime. Table 3.16 describes what configuration leads to the unsafe SO loadings in Table 3.13. For example, the insecure configuration of `DT_RPATH` causes unsafe resolutions that have 140 SO-hijacking directories.

Resolution failure. According to Table 3.13, most of the resolution failures happen in a few applications. Specifically, 95.8% of the resolution failures (*i.e.*, 183/191) are detected from only four

Type	Frequency
Incorrect spec.: prefix and suffix	41
Incorrect spec.: prefix only	29
Incorrect spec.: suffix only	62
Incorrect directory search order	1
Unsupported file	21

Table 3.17: Analysis of resolution failure: filename.

applications.

Our evaluation shows that we detected most of the resolution failures caused by unsafe full-path specification from `Amarok 2.3.0` (*i.e.*, 34/37). The main reason of these failures is that `Amarok 2.3.0` tries to load the libraries provided by applications that are not installed. For example, the application assumes that `RealPlayer 8` is installed on the current system. However, this does not always hold, in which case a resolution failure happens when it tries to load `drvc.so` from `/usr/lib/RealPlayer8/Codecs`. Besides `Amarok 2.3.0`, `Evolution 2.28.3` and `Firefox 3.6.10` try to load `libnssckbi.so` from some particular sub-directories of the current user's home directory. However, the file does not exist in these directories. Similarly, `RealPlayer 11.0.2.1744` tries to load a nonexistent `log.so` file from its plugin directory (*i.e.*, `/opt/real/RealPlayer/plugins`).

Table 3.17 analyzes the filename-based resolution failures. According to our analysis, the detected resolution failures are generally caused for the following reasons: incorrect specification, incorrect directory search order, and unsupported shared library.

Incorrect specification. The filename of the shared library in Linux generally consists of three parts: *prefix*, *soname*, and *suffix*. The prefix is the string 'lib', and the soname is the name of the library. The suffix often consists of a file extension and an optional version number. For example, the filename of the X.Org X11 library, `libX11.so.6`, has the following prefix, soname, and suffix: `lib`, `X11`, and `.so.6`.

To perform safe dynamic loading, it is necessary to correctly specify these three parts of the target file name. However, programming mistakes often happen. In particular, the prefix or the suffix can be incorrect, or even missing, leading to resolution failures. For example, `Skype 2.1` tries to load the X.Org X11 libXcursor runtime library by specifying `Xcursor.so.1`. However, this loading

fails because the correct file name of the library is `libXcursor.so.1` and the specification misses the prefix. Similarly, `Kmail 1.13.2` fails to load the X.Org Xfixes library `libXfixes.so.3`, because it specifies the library name with incorrect suffix. In particular, the correct name of the library on the current system is `libXfixes.so.2`.

Incorrect directory search order. `Pidgin 2.6.6` tries to resolve the `libnssckbi.so` file. To this end, the OS checks the following four directories: `/lib`, `/lib/i486-linux-gnu`, `/usr/lib`, and `/usr/lib/i486-linux-gnu`. The resolution fails, because the file resides in the `/usr/lib/nss` directory on the current system.

Unsupported shared library. Loading shared libraries unsupported by the current system leads to resolution failures. For example, `Brasero 2.30.2` tries to load `libdvdcss.so.2` to access encrypted DVDs, but the file is not included in many Linux distributions [98].

Unsafe resolution. According to Table 3.15, the sub-directories of the two directories, `/opt/*` and `/usr/lib/*`, account for most of SO-hijacking directories (*i.e.*, 973/989). This shows that such unsafe loadings can be hijacked in the directories related to optional application packages such as `Acrobat Reader` and user libraries.

We also analyzed how filename is resolved when unsafe resolution happens. Our analysis shows that most of the filename specifications are determined by cached directories. In particular, we observed that 558 out of 561 filename specifications are resolved by cached directories. Thus, most of the detected unsafe resolutions can be avoided by not specifying `DT_RPATH`, `LD_LIBRARY_PATH`, and `RUNPATH` when loading the cached libraries (*cf.* Table 3.16).

As an example of unsafe resolution for non-cached files, `Evolution 2.28.3` tries to load a calendar component by specifying its filename, `libevolution-calendar.so`. In this case, the component is resolved by checking the directories specified by `DT_RPATH`. In this case, the component is resolved by sequentially checking the directories specified by `DT_RPATH`, which are `/usr/lib/evolution/2.28` and `/usr/lib/evolution/2.28/components`.Because the first directory is checked before the resolution, the loading of the calendar component can be hijacked from this directory.

Browser	Generation (s)	Analysis (ms)
Chrome 6.0.472.63	44	5.68
Firefox 3.6.10	92	14.68
Konqueror 4.4.2	74	22.40
Opera 10.62.6438	46	3.40
Seamonkey 2.0.8	89	33.30

Table 3.18: Execution time for popular Linux browsers.

Severity

Although unsafe SO loadings on Linux platform happen frequently, its severity is relatively low compared to Windows. In particular, root privilege is generally required to hijack loadings. According to Table 3.15, write privileges for most of the SO-hijacking directories are only assigned to the root user. In our evaluation of severity of unsafe loadings on Linux, we assume that a regular user does not have root privilege, which is the typical setup of user privileges on Linux.

During our evaluation, we only detected two SO-hijacking directories with non-root write privilege (*cf.* Table 3.15). For example, `Evolution 2.38.3` fails to load `libnssckbi.so` from the ∼`/.evolution` directory, which can be exploited to hijack the loading.

Comparing to our evaluation on Windows, it is difficult to exploit the detected resolution failures for remote attacks on Linux. This is mainly due to two reasons. First, as we mentioned earlier, most of the SO-hijacking directories are only writable by the root user. It is difficult for attackers to place malicious files in such directories remotely. Second, by default, Linux does not check the current directory to resolve a fullpath specification. Note that the current directory on Windows can be easily written by remote attackers through social-engineering attacks (*cf.* Section 3.4.1).

Performance

To evaluate the performance of our tool, we measured the execution time for detecting unsafe SO loadings of five browsers on Ubuntu 10.04 running on Core2 Duo 2.4GHz processor with 2GB RAM. Table 3.18 shows the execution time of each phase for all analyzed browsers. According to our results, our technique is practical to detect unsafe SO loadings of large, complex software applications, such as web browsers.

3.4.3 Implications of Our Findings

In Sections 3.4.1 and 3.4.2, we have analyzed the prevalence and severity of unsafe loadings on Windows and Linux. This section discusses the implications of our analysis results.

Windows vs. Linux. One interesting discovery is that unsafe SO loadings in some applications on Linux rarely happen. For example, we only detected one resolution failure out of 129 instances of dynamic SO loadings captured from Brasero 2.30.2. Furthermore, we could not detect any unsafe loadings out of 152 dynamic SO loadings by Rhythmbox 0.12.8. The loading safety of the tested Linux applications is mainly due to flexible runtime construction of the directory search order on Linux. In particular, some configurations shown in Table 3.4 to determine the directory search order are optional. For example, if `LD_LIBRARY_PATH` is not specified, the OS determines the search order without considering it. Such flexibility can lead to safe dynamic SO loading. These Linux applications mostly load the cached SO files, and no elements before `CACHED_DIR` in Table 3.4 (such as `LD_LIBRARY_PATH`) affect the directory search order for each loading. In this case, `CACHED_DIR` determines the first search directory to resolve the fullpath of the cached SO file. If `/etc/ld.so.conf` is configured correctly, the resolution of the cached file is safe.

Compared to Linux, Windows adopts a less flexible mechanism to construct the directory search order. Specifically, checking particular directories for component resolution is mandatory (*cf.* Table 3.2). Although Windows supports the *SetDllDirectory-based Search Order*, it is not sufficient. For example, the directory of the application loaded is always checked at first. This inflexibility of Windows causes it to have much more unsafe DLL loadings than Linux. Recall that the application directory serves as the most prevalent type of DLL-hijacking directories (*cf.* Tables 3.9).

Resolution failure vs. unsafe resolution. According to Tables 3.6 and 3.13, unsafe resolution is much more prevalent than resolution failure on both Windows and Linux.

Although resolution failure occurs less frequently, it can lead to remote attacks on Windows. In particular, Windows searches the current directory (*i.e.* ".") to resolve nonexistent components, and the directory can be written from remote sites via social engineering-based attacks (*cf.*, Section 3.4.1). This type of vulnerabilities can be remotely exploited. Although resolution failure also happens on Linux, the SO-hijacking directories shown in Table 3.15 are difficult to be written by remote users.

To exploit the unsafe resolution, attackers should have the write privilege to the DLL- or SO-hijacking directories. However, it is difficult for remote attackers to have such privilege and access the local file system on the victim host.

Filename-based dynamic loading on Windows. Table 3.7 shows that dynamic component loadings using filename specifications are often unsafe on Windows. In particular, 67.6% (1,055/1,560), 70.4% (1,067/1,515) and 75.5% (1,086/1,438) of the filename-based target component loadings are unsafe on Windows XP, Windows Vista, and Windows 7, respectively. The main reason for this is that Windows iterates through predefined sequences of directories to locate a target component when its filename is specified (*cf.*, Table 3.2). This inflexibility makes dynamic loading of system libraries unsafe because the System directory is not the first one checked by the OS. Thus, the first searched directory can be exploited by an attacker to hijack the dynamic loading of a system library (*cf.*, Tables 3.9 and 3.10). Note that unsafe resolutions can be significantly reduced by specifying the fullpaths of the target components. In particular, developers can secure all filename-based component loadings in Table 3.7 by specifying the target components' fullpaths.

OS flaws vs. application flaws. Resolution failures on both platforms are mainly caused by programming errors. In particular, developers may load a target component without checking whether it exists on the current system. However, the main reasons for having unsafe resolutions differ on each platform.

Windows. According to Table 3.7, unsafe loadings happen in all the test subjects on Windows, among which unsafe resolutions are especially common. As we mention earlier, their main cause is the inflexible resolution mechanism on Windows. Although Microsoft supports mechanisms such as side-by-side assembly [133] to control the directory search orders, they are not adopted by default. Also, Windows performs chained loading based on filename specifications [106]. This insecure OS-level mechanism can make chained loadings unsafe. For example, Table 3.7 shows that 56.0% (537/958), 58.3% (547/938), and 64.0% (480/750) of the total chained loadings are unsafe on Windows XP, Windows Vista, and Windows 7, respectively.

Linux. Section 3.3.2 shows that, on Linux, developers can control the directory search order as needed. Thus, we can consider any unsafe resolution to be a flaw in the application. According to Table 3.14, unsafe resolutions are not very common in Linux-based applications. For example,

Chrome has no unsafe loadings on Linux, while it has many unsafe loadings on Windows (*cf.*, Tables 3.6 and 3.13).

Privilege assumption on Windows. Our evaluation on Windows assumes that users have the administrative privilege. If this assumption does not hold, the severity of unsafe loadings is significantly reduced. In particular, all the local attacks exploiting unsafe resolutions can be prevented. For example, Table 3.9 shows that only administrators can write to the DLL-hijacking directories. However, remote attacks are still feasible because malicious files sent by an attacker are generally stored in the directories writable by the current normal user. For example, suppose that a user receives a malicious archive file from a "Document with Component" attack. The user should have the write privilege to the directory that stores the file.

3.4.4 Comparison to Related Work

Our tool detects unsafe component loadings from runtime traces (*cf.*, Figure 3.1). Thus, the detection results depend on code coverage of the captured traces. In particular, our approach detects unsafe loadings from the code covered by the traces. This limited code coverage is the standard limitation of dynamic analysis. To evaluate our tool's completeness, we compare our technique with a recently-released tool that detects unsafe DLL loadings [50].

Moore's Approach

Although our earlier work [94] is the first to automatically detect unsafe component loadings and demonstrate their prevalence and security implications, there are two related recent efforts. In August 2010, Moore and Acros Security announced that unsafe DLL loadings are prevalent and can lead to remote code execution [148, 166]. They referred to unsafe loadings as "DLL Preloading" and "Binary Plating", and came to the same conclusion as in our earlier published work [94]. We disclosed our work and results to Microsoft in August 2009 and issued a technical report in January 2010.

Moore released a tool to detect unsafe DLL loadings in August 2010 [50]. The tool works in a few phases: 1) test case generation, 2) file system access monitoring, and 3) exploitation check. In order to generate the test cases, the tool creates text files whose extensions are known by the OS.

Application	Approach	Test cases	Failed			Unsafe		Remote attacks	
			Fullpath	Filename		Filename			
			T	T	C	T	C	Shortcut	Document
iTunes 10.0.1	Ours	mp3 file	0	1	0	38	46	0	0
	Moore's	text file with the mp3 extension	0	0	0	0	0	0	0
Media Player 12	Ours	wmv file	0	1	0	31	7	0	0
	Moore's	text file with the mp3 extension	0	0	0	0	0	0	0
		text file with the asx extension	0	0	0	0	0	0	0
		text file with the wmv extension	0	0	0	0	0	0	0
Quicktime 7.6.8	Ours	mp3 file	0	0	0	27	30	0	0
	Moore's	text file with the mp3 extension	0	0	0	0	0	0	0
RealPlayer SP 1.1.5	Ours	rm file	2	3	0	26	20	3	0
	Moore's	text file with the rm extension	0	0	0	0	0	0	0
		text file with the amr extension	0	1	0	0	0	0	0
		text file with the awb extension	0	1	0	0	0	0	0
		text file with the divx extension	0	2	0	0	0	0	1
Winamp 5.58	Ours	mp3 file	4	1	0	12	9	0	0
	Moore's	text file with the mp3 extension	0	0	0	0	0	0	0
		text file with the asx extension	0	3	0	0	0	0	1
		text file with the b4s extension	0	3	0	0	0	0	1

Table 3.19: Comparison between our approach and Moore's one.

For example, a text file with the `.rm` extension serves as a test case for RealPlayer. The generated test cases determine the test subjects by choosing the applications associated with the corresponding extensions. Afterwards, the tool detects resolution failures by monitoring runtime file system accesses of each test subject. Once all the test subjects are analyzed, the tool checks whether or not the detected resolution failures can lead to remote attacks. To this end, it adopts an approach similar to ours for detecting the "Document with Component" attack (*cf.* Section 3.4.1).

Our Approach vs. Moore's One

To evaluate our technique's relative completeness, we detect unsafe loadings of the five multimedia players using our and Moore's tools on Windows 7 and compare their detection results. Table 3.19 shows the detailed comparison.

As mentioned earlier, our tool can detect both types of unsafe component loadings, while Moore's tool focuses on the detection of resolution failures by monitoring file system accesses performed by the test subjects. For example, our tool detects 46 unsafe resolutions from RealPlayer SP 1.1.5, while Moore's tool does not detect any unsafe resolutions.

According to Table 3.19, both approaches are not complete. In particular, our tool can detect unsafe loadings missed by Moore's tool (and vice versa). For example, only our tool detects the

security vulnerabilities in RealPlayer that can be exploited by "Shortcut with Component" attacks. On the other hand, Moore's tool detects two unsafe loadings in Winamp that can lead to "Document with Component" attacks, but our tool does not.

The main reason for our tool's incompleteness is the selection of test cases. For example, our tool can also detect the unsafe DLL loadings of Winamp when we use the test cases generated by Moore's tool. In addition, both approaches may miss unsafe loadings in code that is not exercised by the selected test cases. To mitigate this issue, one needs to develop techniques to achieve better code coverage (*e.g.*, test case generation or static analysis), which we leave for future work.

3.5 Mitigation Techniques

This section discusses general and platform-specific techniques to mitigate unsafe component loadings.

3.5.1 General Techniques

Use fullpath. Because the filename specification resolves the target component by iterating through the directories, it may lead to unsafe resolution. This problem can be solved by specifying the target component based on its full path, because the fullpath specification determines its target file directly without iteratively searching a set of directories. In order to generate correct fullpath specifications, system calls that return full paths of the target directories can be used. For example, suppose a developer wants to load a DLL in the system directory at runtime on Microsoft Windows. In this case, `GetSystemDirectory` function can be used to determine the full path of the DLL. In particular, after obtaining the path of the system directory through the system call, the developer can concatenate the path with the filename of target DLL to obtain its full path. For instance, if a developer wants to load `WS2HELP.DLL` in the system directory, safe DLL resolution can be achieved by concatenating `WS2HELP.DLL` with the system directory path obtained by the `GetSystemDirectory` function (*i.e.*, `C:\Windows\System32`).

Resolve system call at runtime. According to Section 3.4, chained loading of components also causes unsafe resolution. This can be mitigated by resolving system calls at runtime as much as possible. In particular, if we resolve the address of the target system call exported by a compo-

nent and invoke it at runtime, the component file is not considered a dependent component and is not loaded at load-time. For example, suppose we want to invoke the `db_create` function of `libdb.so.3`, we can obtain the function's address by using the `dlopen` and `dlsym` functions at runtime, and invoke the target function based on this address. Note that this mechanism makes software safer but less efficient.

Confirm file existence. As we mentioned in Section 3.4, resolution failures can cause serious security vulnerabilities in software. The main reason is that many programs make the false assumption that the target component exists in the system. Therefore, to avoid resolution failures, it is important to check existence of the target files before loading them.

Check validity of loaded components. Because a program resolves a target component based on its name, it is difficult to determine whether the resolved component is the file intended by the program. To address this problem, application developers can provide the signature of the target file to determine the validity of the loaded component. For example, the hash value and the RSA signature can be used for validation. However, this approach makes software less flexible. Also, malicious users can reverse engineer the validity check and bypass it. Therefore, OS-level protection mechanisms are necessary to adopt this mitigation technique.

Check current OS version. As we discussed in Section 3.4, a set of system libraries depends on the version of the operating system. Because many applications are developed to be executed under different platforms, they should check the version of the OS and load only the supported components.

Provide tools for checking third-party components. Unsafe component loadings performed by third-party components can lead to serious security holes in the applications hosting them. Because of this issue, although the applications resolve the components safely, they can be attacked by exploiting vulnerabilities in the third-party components. To mitigate this problem, it is necessary for application developers to provide the developers of the third-party components with tools to check the safety of their components.

3.5.2 Windows-specific Techniques

Use SetDllDirectory function. As we mentioned in Section 3.4.1, the current directory at the point of a resolution failure may cause remote code execution attacks. To mitigate this type of attacks, we can use the `SetDllDirectory` function which can add an arbitrary directory instead of the current directory. Especially, this function can remove the current directory from the directory search order. This approach can effectively block remote code execution attacks discussed in Section 2.1.3. In particular, Microsoft adopts this approach to fix the blended attack combined with the Safari's Carpet Bomb attack [114].

Install applications in the admin-writable directory. According to Table 3.9, the application directories are the most vulnerable ones to unsafe resolution. Therefore, unsafe resolutions performed by non-admin users can be significantly reduced by installing applications in directories only writable by administrators (*e.g.*, the `Program Files` directory on Microsoft Windows).

Do not disable User Account Control (UAC). Microsoft Windows platforms have supported User Account Control (UAC) [149] since Windows Vista to prompt the confirmation dialogs whenever users perform security-related tasks. This UAC feature prevents the unintended copy of arbitrary files to particular DLL-hijacking directories such as the `Program Files`, which mitigates the exploitation of unsafe DLL loadings. However, many users have complained that the UAC dialogs frequently show up [146], and thus have disabled UAC. This unsafe setting makes users vulnerable to the attacks exploiting unsafe component loadings. Therefore, it is necessary not to disable the UAC feature to mitigate such attacks.

3.5.3 Linux-specific Techniques

Configure safe directory search order. As we mentioned in Section 3.4.2, Linux provides a flexible mechanism to configure the directory search order at runtime. In particular, the attributes in Table 3.4 determine the directory search order. Thus, it is possible for programmers to configure the safe directory search order such that the first checked directory contains the target file.

Cache SO files. The fullpath of the SO file on Linux can be cached by setting the configuration file `/etc/ld.so.conf`. This feature allows us to specify the intended fullpath for any filename specification. Thus, if we cache the target file without specifying any attributes before `CACHED_DIR`

in Table 3.4, the fullpath of the file can be safely resolved. Note that we could not detect any unsafe SO loading from some of our test subjects adopting this technique (*cf.* Section 3.4.2).

3.6 Related Work

We survey additional related work besides those on detecting unsafe DLL loadings discussed in Section 3.4.4. We divide the related work into four categories: safe component resolution, safety improvement of browser plugins, vulnerability analysis and detection, and non-control-data attacks.

Safe component resolution. Chari *et al.* [31] presents a mechanism, `safe-open`, to prevent unsafe component resolutions in Unix by detecting modifications to path names by untrusted users on the system. In comparison, we propose a dynamic analysis to discover unsafe component loading vulnerabilities in the software itself.

Safety improvement of browser plugins. Secure browsers [59,61,62,153] have been introduced to mitigate risks caused by unsafe usage of third-party plug-ins. Gazelle [153] and OP [62] browsers adopt OS-level sandboxing techniques to reduce damages introduced by unsafe plugin usage. Internet Explorer utilizes a kill-bit [84] to prevent malicious ActiveX components from being loaded. Grier *et al.* [61] propose security policies for secure plugin execution. These techniques aim at providing software platforms with secure plugin usage, while our technique aims at detecting unsafe loadings of general software components.

Vulnerability analysis and detection. Testing and analysis techniques for detecting software vulnerabilities have been well explored. Most of the previous approaches have focused on detecting low-level, unexpected program behaviors such as memory corruption errors [28, 29, 42, 57, 95, 127, 129, 163] and integer overflows [26, 112, 154]. Although these approaches have shown promising results in detecting such vulnerabilities, none has targeted the detection of unsafe component loadings; our work formulates the problem and introduces the first effective automated technique to detect such vulnerabilities.

Non-control-data attack. Unsafe component loading can also be considered an example of non-control-data attacks because it does not alter the control data of the target program. Chen *et al.* [32] surveyed attack techniques that corrupt application data, which includes user identity data, configuration data, user input data, and decision-making data, and presented a detailed analysis and de-

fense mechanism. Compared to those non-control-data attacks, unsafe dynamic loading is mainly due to defects in the component loading procedure, while they are originated from unsafe handling of application data. In addition, the attack vectors are different. In particular, unsafe component loading can be exploited by placing malicious files in the component-hijacking directories, while non-control-data attacks corrupt certain application data to exploit unsafe processing of the data.

3.7 Conclusions and Future Work

In this chapter, we have described the first analysis technique to detect unsafe dynamic component loadings. Our technique works in two phases. It first generates profiles to record a sequence of component loading behaviors at runtime using dynamic binary instrumentation. It then analyzes the profiles to detect two types of unsafe component loadings: resolution failures and unsafe resolutions. To evaluate our technique, we implemented tools to detect unsafe component loadings on Microsoft Windows and Linux. Our evaluation shows that unsafe component loadings are prevalent on both platforms and more severe on Windows platforms from a security perspective. In particular, our tool detected more than 4,000 unsafe component loadings in popular software on both platforms. It also discovered 41 potential remote code execution attacks on Microsoft Windows.

For future work, we are interested in developing static binary analysis techniques to detect unsafe component loadings. Although our dynamic analysis is effective, it may suffer from the standard limitation of dynamic analysis, namely the code coverage problem. Specifically, our approach may miss unsafe component loadings that can happen. We plan to develop sound, practical static analysis techniques to complement the dynamic analysis we introduced here.

Chapter 4

Static Detection of Unsafe Component Loadings

4.1 Introduction

Dynamic loading of software components is a commonly used mechanism to achieve better flexibility and modularity in software. For an application's runtime safety, it is important for the application to load only its intended components. However, programming mistakes may lead to failures to load a component, or even worse, to load a malicious component. The proposed dynamic technique in Chapter 3 has shown that these errors are both prevalent and severe, sometimes leading to remote code execution attacks. The work is based on dynamic analysis by monitoring and analyzing runtime component loadings. Although simple and effective in detecting real errors, it suffers from limited code coverage and may miss important vulnerabilities. Thus, it is desirable to develop effective techniques to detect *all possible* unsafe component loadings.

Although the proposed dynamic technique is effective at detecting real unsafe loadings, it may miss errors because of limited code coverage, an inherent weakness of dynamic analysis. We illustrate this issue using *delayed loading*, an optimization to postpone the loading of infrequently used components until their first use. Delayed loading is challenging for dynamic detection because it is difficult to trigger all delayed loadings at runtime. Figure 4.1 shows a code snippet that uses delayed loading in Microsoft Windows. The code shows two functions `f1` and `f2` that use components regis-

```
 1  void f1() {
 2      ...
 3      pDelayDesc1 = &WINSPOOL_DRV_DelayDesc;
 4      // WINSPOOL_DRV_DelayDesc.dllname = "WINSPOOL.DRV"
 5      func_addr = __delayLoadHelper2(
 6         pDelayDesc1, "OpenPrinter"
 7      );
 8      ...
 9  }
10  void f2() {
11      ...
12      pDelayDesc2 = &COMDLG32_DLL_DelayDesc;
13      // COMDLG32_DLL_DelayDesc.dllname = "COMDLG32.DRV"
14      func_addr = __delayLoadHelper2(
15         pDelayDesc2, "GetSaveFile"
16      );
17      ...
18  }
19  int __delayLoadHelper2(pImgDelayDesc, funcName) {
20      hMod = pImgDelayDesc->hMod; // init value = 0
21      if (hMod == 0) {
22         target_dllname = pImgDelayDesc->dllname;
23         hMod = LoadLibrary(target_dllname);
24         pImgDelayDesc->hMod = hMod;
25      }
26      func_addr = GetProcAddress(hMod, funcName);
27      return func_addr;
28  }
```

Figure 4.1: Motivating example.

tered for delayed loading. In particular, `f1` and `f2` retrieve the addresses of `OpenPrinter` exported by `WINSPOOL.DRV` and `GetSaveFile` exported by `COMDLG32.DLL` respectively. Although the example only shows two functions `f1` and `f2`, in practice, there are often many more. The infrequent use of the components makes it difficult, if not impossible, to trigger all possible loadings at runtime. Although we have illustrated the problem using delayed loading, poor coverage of dynamic analysis is a general concern for detecting unsafe loadings, as our results also confirm (*cf.* Section 4.4).

In this chapter, we present the first *static* analysis to detect unsafe loadings from program binaries. Two pieces of essential information are needed: 1) all components that may be loaded at each loading call site, and 2) the safety of each possible loading. While the second part is straightforward, the key challenge lies in the first part—how to precisely and scalably compute the possible loadings. Our *key observation* is: for a given invocation of the loading system call, the set of possible loaded components is determined by the system call's parameter values, which are often determined through computations that originate not far from the call site. From these observations, we design

a two-phase analysis: *extraction* and *checking*. The extraction phase is *demand-driven*, working backward from each loading call site to compute the set of possible loadings; the checking phase determines the safety of a loading by examining the relevant directory search order at the call site.

Context-Sensitive Emulation. To realize the backward computation of parameter values during the extraction phase, we introduce *context-sensitive emulation*, a novel combination of slicing and emulation. For a given call site, we extract its context-sensitive *executable* slices w.r.t. its parameters, one for each execution context. We then emulate the slices to compute the parameter values.

Incremental and Modular Slicing. One technical obstacle is how to compute backward slices scalably. Standard slicing techniques [5, 23, 66, 119, 124, 134] are based on computing a program's complete system dependence graph (SDG) *a priori* and are thus limited in scalability. Because we only need to consider loading call sites and the execution paths to compute the parameter values to the calls are usually relatively short, only a small fraction of the complete SDG is relevant for our analysis. This motivates the use of an *incremental* and *modular* slicing algorithm (*cf.* Section 4.3)—incremental because we build the slices lazily when necessary; modular because when we encounter a function call `foo(x,y)`, we use an inferred summary of what dependencies `foo`'s parameters and return value have in analyzing the caller. At the end, we connect the function-level slices in the standard way by linking formal and actual parameters.

Emulation of Context-sensitive Slices. Once we have computed the backward slice s w.r.t. a given loading call site, we need to compute possible values for the relevant parameters. One natural solution is to perform standard symbolic analysis on the slice to compute the values. The main challenge for this approach is the difficulty in reasoning symbolically about system calls because the relevant parameters often depend on complex, low-level system calls. For example, many Windows applications invoke the system call `RegQueryValueExW` to retrieve the fullpath of the target specification stored in the registry key. The system call invokes more than 100 distinct system calls exported by five libraries. To symbolically analyze the system call, it is necessary to symbolically execute its invoked system calls as well, leading to path explosion. Thus, it is difficult in practice to engineer and scale symbolic analysis to compute the possible values of the parameters.

To overcome this difficulty, we use emulation. In particular, we generate, from the backward slice s, a set of *context-sensitive executable sub-slices*, which we then *emulate* to compute the pa-

rameter values (*cf.* Section 4.3). Essentially, we inline callees' function-level slices in each execution context to produce s's sub-slices $s_1, \ldots, s_n$. Instructions in each sub-slice s_i are next emulated topologically, respecting their data- and control-flow dependencies.

For evaluation, we implemented our technique in a prototype tool for Windows applications. We evaluated our tool's effectiveness against the dynamic technique in terms of precision, scalability, and coverage. Results on nine popular applications show that our tool is precise and scalable (*cf.* Section 4.4). For example, it took less than two minutes to analyze each of the nine test subjects, including large applications such as Acrobat Reader, Quicktime, and Safari. The results also show that our proposed context-sensitive emulation achieves orders of magnitude reduction in the size of the code needed to be analyzed and crucially contributes to the scalability of our technique. In terms of coverage, our tool detected many more possible unsafe loadings and nicely complements the dynamic technique.

Main Contributions:

- We have developed the first static binary analysis to detect unsafe component loadings. Because of its scalability and higher code coverage, our technique effectively complements the existing dynamic technique.

- We have proposed context-sensitive emulation, an effective approach that combines slicing and emulation for the precise and scalable analysis of runtime values of program variables.

- We have implemented our technique and evaluated its effectiveness by detecting unsafe loadings in nine popular Windows applications.

The rest of this chapter is organized as follows. Section 4.2 illustrates our technique with a running example. Section 4.3 presents a detailed description of our static detection algorithm. We describe our implementation and evaluation in Section 4.4. Finally, Section 4.5 surveys additional related work, and Section 4.6 concludes with a discussion of future work.

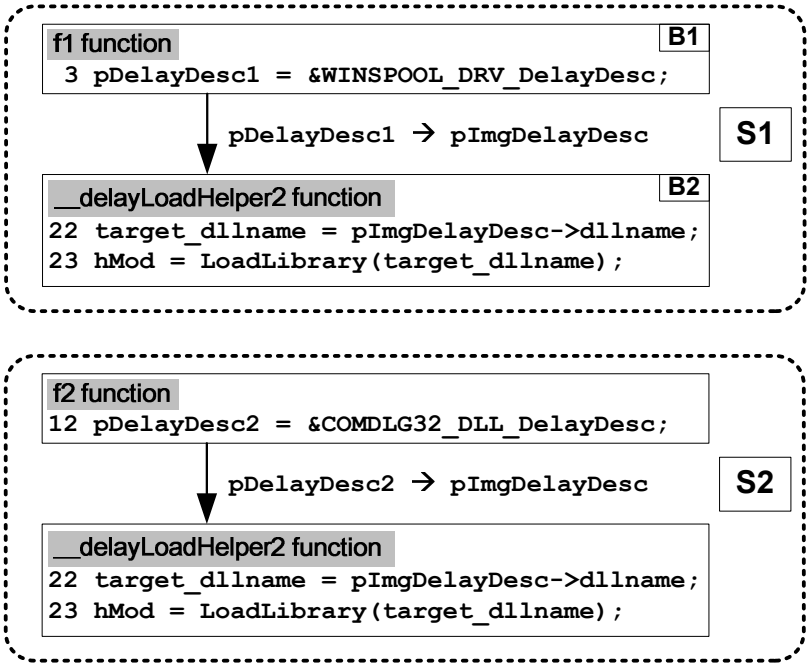


Figure 4.2: Example context-sensitive backward slices.

4.2 Overview

This section illustrates our technique with the example shown in Figure 4.1. Our technique works on binaries, but for presentational purposes, we show the example in C-like pseudo code.

Extraction Phase. We first identify call sites for component loading. In the example, line 23 corresponds to a call site because of the `LoadLibrary` system call. The system call's only parameter `target_dllname` determines which component should be loaded. We use context-sensitive emulation to compute its possible values.

Incremental and Modular Slicing. We start with the call site on line 23 as the slicing criterion and extract its caller's function-level slice. Program slicing generally considers data and control flow dependencies to extract a slice. In our setting, since the main goal is to compute possible values of `target_dllname`, we focus on data dependencies and produce the slice: lines 22 and 23. To compute the possible values of `target_dllname`, we need to extract the code that computes `pImgDelayDesc`, the first parameter of the `__delayLoadHelper2` function. To this end, we continue the backward slicing w.r.t. a new slicing criterion, which is determined based on caller-callee relationship and the callee's function prototype. In our example, there exist two call sites

for __delayLoadHelper2, lines 5–7 in f1 and lines 14–16 in f2. Thus, we continue with two instances of intraprocedural backward slicing w.r.t. two new slicing criteria: pDelayDesc1 on line 6 and pDelayDesc2 on line 15. Two slices are generated for f1 and f2. We produce two context-sensitive interprocedural slices by instantiating twice the slice for __delayLoadHelper2 and linking each instance with its respective caller's slice. We also maintain the mapping between each of the new slicing criteria and the callee's corresponding parameters for the later emulation phase. Because neither f1 nor f2 takes any input, we terminate the slicing computation. Figure 4.2 displays the two computed context-sensitive backward slices w.r.t. target_dllname.

Emulation of Context-sensitive Slices. To compute possible values for target_dllname, we emulate the two slices in Figure 4.2. We need to schedule the instructions in the slices before they can be emulated. We do so respecting the data and control flow dependencies among the instructions. Specifically, we first schedule the basic blocks in topological order with respect to the data flow dependencies among them. We then determine the ordering of the instructions in each scheduled basic block in terms of their ordering in the original code. More concretely, if i_1 precedes i_2 in the original code, we emulate i_1 before i_2.

For example, the instructions in the first slice in Figure 4.2 are scheduled as follows: 1) two basic blocks, B1 and B2, contain the instructions of the slice: B1 for line 3 in f1 and B2 for lines 22 and 23 in __delayLoadHelper2; 2) B2 depends on B1 because pImgDelayDesc used by B2 on line 22 is initialized on line 3 in B1, leading to the following scheduling: B1→B2; 3) we schedule the instructions of each basic block in terms of their control flow dependencies: lines 3, 22, and 23. For parameter passing, we initialize the formal parameter with the corresponding actual parameter's value. In our example, the value of the formal parameter pImgDelayDesc of __delayLoadHelper2 is provided by f1 through the value of pDelayDesc1.

After successfully emulating B1 and B2 for the first slice, we obtain the possible value of target_dllname: "WINSPOOL.DRV". Similarly, we obtain its other possible value after emulating the second slice: "COMDLG32.DLL".

Checking Phase. In our example, "WINSPOOL.DRV" and "COMDLG32.DLL" are potentially loaded at runtime. When the OS loads these components, it iterates through a sequence of directories, determined at runtime, to locate the specified files. In this case, these loadings are unsafe, if the OS

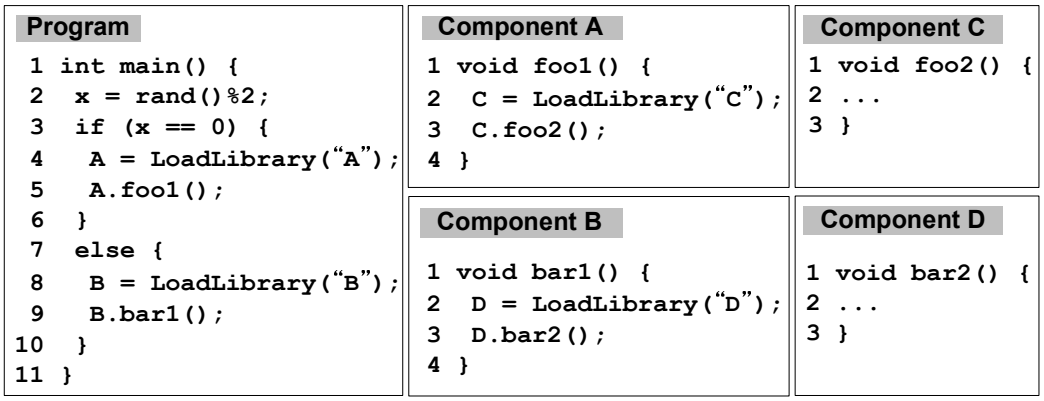


Figure 4.3: Component-integrating code.

checks multiple directories to resolve these components on default configuration. This is because these loadings can be hijacked by placing an arbitrary file named `WINSPOOL.DRV` or `COMDLG32.DLL` in the directories checked before the intended resolution. We check whether or not the specified files exist in the first directory searched. Because Microsoft Windows searches first in the directory where the program is installed [45], the loadings for these two components are unsafe if they do not exist in the program directory.

4.3 Static Detection Algorithm

We now present the details of our analysis. Our technique statically detects unsafe component loadings to achieve high coverage. It first extracts the target component specifications from possible code region executed at runtime and check their safety based on Definition 2.1.6.

The executed code region is determined by loaded components. Figure 4.3 depicts the component loading code whose execution path controlled by a random variable `x`. If `x` is zero, `foo1` of component `A` and `foo2` of component `C` are executed. Otherwise, `bar1` of component `B` and `bar2` of component `D` are executed. Our observation is that each execution path covers the partial code region of the loaded components. For example, if `x` is zero, the partial code regions of components `Program`, `A`, and `C` are executed. From these observations, we design our static detection as shown in Figure 4.4: *extraction* and *checking*. From the extraction phase, we obtain a set of the target component specifications from the components that can be loaded at runtime. In the checking phase, we evaluate the safety of each target specification based on Definition 2.1.6.

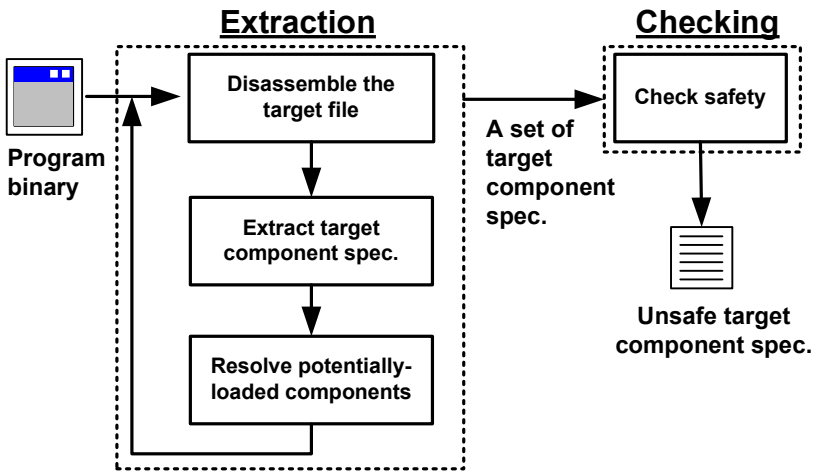


Figure 4.4: Detection framework.

4.3.1 Extraction Phase

A component can load other components at loadtime or runtime. This loading introduces *load-time* and *runtime* dependencies among components [4]. Based on these dependencies, we determine components that can be loaded during program execution. Specifically, we recursively resolve the components from the program file based on their loadtime and runtime dependencies. To resolve the dependent components, the corresponding target specifications, *i.e.*, full path or file name, are needed. For loadtime dependencies, compilers specify the dependent components in the executable format. For example, the names of the loadtime dependent components are stored in `IMAGE_IMPORT_DIRECTORY` with the PE format [106]. To obtain the specifications of the runtime dependent components, we compute values of parameters to component-loading system calls. This suffices for our setting because program dynamically loads components via the system calls and their parameters determine the loaded components.

As an example of recursive resolution, we search the components that are potentially loaded by `Program` in Figure 4.3. Suppose that components `E` and `F`, which have no loadtime and runtime dependent component, implement the `rand` and `LoadLibrary` functions, respectively. In this case, `Program` loads components `E` and `F` on its startup. Regarding runtime dependencies, `Program` dynamically loads components with the specifications, `"A"` on line 4 and `"B"` on line 8. From this information, we can detect the potentially-loaded components by simulating component resolution. Similarly, we can infer that `C`, `D` and `F`, which are loaded by `A` and `B`. Because `C` and `D` have no

```
1   PUSH   EAX
2   PUSH   EAX
3   PUSH   offset 0x7D61AC5C; "xpsp2res.dll"
4   CALL   DWORD PTR DS:[LoadLibraryExA]
```

(a) Memory indirect.

```
1   MOV    EBX, DWORD PTR DS:[LoadLibraryW]
2   PUSH   offset 0x65015728; "CABINET.DLL"
3   CALL   EBX
```

(b) Register indirect.

Figure 4.5: Two types of component-loading call sites.

loadtime and runtime dependent components, we stop the resolution process. Thus, we detect the seven components potentially loaded at runtime: `Program`, `A`, `B`, `C`, `D`, `E`, and `F`.

The key step of the extraction phase is to obtain the target specification for component loading in a binary. The specification of a loadtime dependent component can be easily obtained from the binary file format. However, extracting the specification of a runtime dependent component is nontrivial because it often requires to locate the code relevant to the value of the specification and analyze its execution. For example, the target component specification for system libraries under Microsoft Windows is sometimes determined by concatenating the system directory path and the file name. To obtain the specification, it is necessary to extract the related code and analyze its execution result.

The concrete value of the parameter to the component-loading system call serves as the specification for the runtime dependent component. From this observation, we extract the specification by searching for the program variable for the specification and then computing its value via *context-sensitive emulation*, a novel combination of backward slicing and emulation. We describe details of the extraction in the following sections.

4.3.2 Searching Program Variable for Specification

In binary code, invoking the component-loading system calls follows the `stdcall` calling convention [161]. When parameters are passed to the call site, they are pushed from right to left. For example, Figure 4.5(a) represents the binary code corresponding to `LoadLibraryExA(0x7D61AC5C, EAX, EAX)`. Based on the parameter passing mechanism, we locate the program variable, *e.g.*, a register or a memory chunk, which stores the target specification. In particular, we detect the call

site for component loading via static taint data analysis and then extract the input operands of the instructions passing the parameter to the call site. We describe details of each step in the rest of this section.

Locating Component-loading Call Sites. In this phase, we aim at finding the call site for component loading in a binary. Our observation is that software stores the address of the system call implementation in its memory space and utilizes it in the call sites for component loading at runtime. Figure 4.5 shows the two types of component-loading call sites in a binary, which are *memory indirect* and *register indirect*. The main difference between them is what type of the program variable stores the address of the component-loading system call at the call site. While the memory indirect type stores the address in a memory chunk, the register indirect type stores the address in a register, *e.g.*, line 4 in Figure 4.5(a) and line 3 in Figure 4.5(b).

Based on this observation, we locate the component-loading call sites through static taint data analysis. In particular, we define the taint sources and the taint sinks as follows:

- *Taint source*: an instruction that references a memory chunk that stores the address of the component-loading system call.
- *Taint sink*: a branch instruction, *e.g.*, `call`, whose target address is tainted. We consider the taint sink instructions as the call sites.

We now present examples on how to detect call sites. In Figure 4.5(a), line 4 serves as not only the taint source but also the taint sink, *i.e.*, the component-loading call site, because it is the branch instruction, accessing a memory chunk that stores the address of `LoadLibraryExA`. For Figure 4.5(b), line 1 is the taint source, accessing the address of the `LoadLibraryA`, and line 3 is the taint sink, because it is the `call` instruction whose target is the address, stored in `EBX`.

Extracting Parameter Variables. Once a call site is located, we extract the program variables for the target specification from the predefined number of the instructions to pass the parameters to the call site. In particular, we detect the instructions, *e.g.*, `PUSH`, to initialize the top of stack backward from the call site. Because the number of parameters of a component-loading system call is known, we can precisely extract all the variables to define this target specification. For example, the call site in Figure 4.5(a) invokes `LoadLibraryExA`, and it has three parameters, *i.e.*, `0x7D61AC5C`, `EAX`, and `EAX`, via the instructions on lines 1–3.

4.3.3 Context-sensitive Emulation

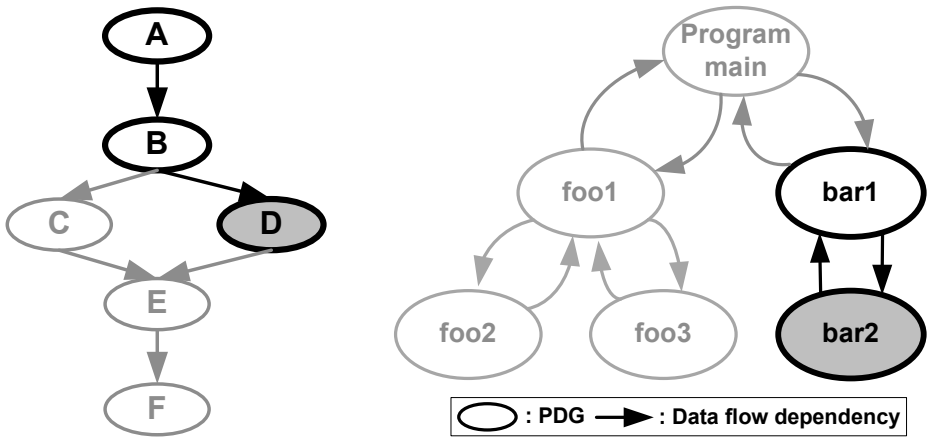
In this phase, we compute the concrete values of the parameter variables extracted in Section 4.3.2. The computation may seem trivial at first. For example, the memory chunk at `0x7D61AC5C` in Figure 4.5(a) contains the target specification, `"xpsp2res.dll"`. However, the computation is in fact challenging because it is necessary to extract the code to compute the variable, requiring interprocedural data flow analyses (*cf.* Figure 4.1). Also, we need the runtime information of the code to obtain the concrete values of the variable. Symbolic analysis can serve as a potential solution. However, symbolic analysis suffers from poor scalability and is limited in handling system calls, which are often complex.

To address this problem, we introduce *context-sensitive emulation*, which novely combines backward slicing and emulation. Based on this combination, we can scalably and precisely compute the values of the variables of interest. We describe its details in the rest of this section.

Backward Slicing. This phase performs the interprocedural backward slicing w.r.t. the parameter variable, extracting the instructions to compute the variable. This problem has been extensively studied, and many slicing algorithms [5,23,66,119,124,134] have been proposed. These algorithms commonly solve the graph reachability problem over a System Dependence Graph (SDG) [66], a set of Program Dependence Graphs (PDGs) [51] and edges capturing data flow dependencies among them. In particular, a SDG is constructed beforehand based on an exhaustive data flow analysis over the subject program. Then, the slicing outcome is determined by traversing the SDG from the given slicing criteria. Although the approach has been widely used, it is not appropriate for our problem setting. The reason is that binary files are generally composed of a large number of instructions, and an exhaustive data flow analysis over all the instructions is very expensive, leading to limited scalability.

Our key observation is that the parameter values are often locally determined, that is the execution paths to compute the variables are relatively short. Thus, exhaustive data flow analysis is not be necessary to extract backward slices w.r.t. the given slicing criteria. Figure 4.6 shows the examples of the unnecessary data flow analysis during intraprocedural and interprocedural backward slicing.

Figure 4.6(a) shows an example of the CFG for constructing the PDG. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D`. In this case, the bold instructions often only



(a) Intra-backward. (b) Inter-backward.

Figure 4.6: Unnecessary data flow analysis.

affect the instruction `D` in terms of control flow. It is possible that the instruction `D` can be affected by the instructions without control flow dependencies. For example, the instruction `E` initializes a variable and the instruction `B` reads it. However, this case rarely happens in our problem setting in practice, because the parameters for the specification are generally computed by the instructions executed before the component-loading call sites.

Suppose that Figure 4.6(b) depicts the SDG for the interprocedural backward slicing. If the instructions of the bold PDGs for `bar1` and `bar2` are only traversed during slicing, is it not necessary to perform data flow analysis on the instructions of the grayed PDGs. Because the SDG consists of a large number of PDGs in binary and the target specifications are often locally determined, most of the PDGs are not relevant for interprocedural backward slicing w.r.t. the parameter variables for the target specifications.

Based on this insight, we design our slicing technique as demand-driven, reducing the unnecessary analysis of data flow dependencies. In particular, we perform interprocedural backward slicing by incrementally combining the intraprocedural backward slices whose slicing criteria are determined when necessary.

Intraprocedural backward Slicing. For each intraprocedural backward slicing, we analyze only the data flow dependencies among the instructions that are control dependent on the given slicing criteria. To this end, we construct the PDG based on the *predecessor subgraph* w.r.t. the slicing criterion under the CFG. Thus, we can avoid the analysis of the data flow dependency among the

instructions not traversed during slicing. Suppose that we perform intraprocedural backward slicing w.r.t. the instruction `D` in the CFG shown in Figure 4.6(a). If we construct the PDG based on the CFG, the data flow dependencies among all the instructions in the CFG are analyzed. However, the grayed instructions do not affect the instruction `D` in terms of control flow dependencies. By constructing the PDG based on the subgraph composed of the bold instructions, *i.e.*, the predecessor subgraph w.r.t. the instruction `D`, we can avoid some unnecessary data flow analysis when performing slicing.

One challenge for PDG construction is caused by the call site instructions. Because functions are not generally monolithic, it is necessary to identify which call sites affect the slicing criteria. Although traversing the SDG provides such information, it requires the computation of significant amount of unnecessary data-flow dependencies (*cf.* Figure 4.6(b)). To address this problem, we utilize the prototypes of the functions invoked at the call sites. Specifically, we consider a call site instruction as a non-branching instruction during our PDG construction, and analyze the data flow dependencies related to the call site in terms of the prototype of the callee function. For example, a call site invokes a function `foo` whose prototype is `int foo(in,inout)`. In this case, the `foo` is considered an instruction that uses the first/second parameters and defines the second parameter and the return variable. Based on this information, we can effectively determine the data flow dependencies between the call site instructions and the slicing criteria without a whole SDG traversal.

Interprocedural backward Slicing. As aforementioned, an exhaustive SDG construction often leads to significant amount of the unnecessary data flow analysis for interprocedural backward slicing. To address this problem, we construct the interprocedural backward slices incrementally combining the intraprocedural backward slices whose slicing criteria are chosen in a demand-driven manner.

There are two key challenges for this demand-driven combination. First, it is necessary to determine the new slicing criteria if the interprocedural backward slice consists of multiple intraprocedural backward slices. For example, we construct the interprocedural backward slice in Figure 4.6(b) by combining the two intra-backward slices extracted from functions `bar1` and `bar2`. In this case, we need to determine the new slicing criteria in the `bar1` function. Second, the composed interprocedural backward slice needs to be easily handled for the later emulation phase.

Our basic idea for building the new slicing criteria is that the interprocedural data flow dependencies are captured by parameter passing. In SDG-based slicing, the PDGs are connected using the edges that model parameter passing, which are traversed to analyze the dependencies. Based on this idea, we choose the slicing criteria as follows. Suppose that an intraprocedural backward slice s is extracted from an instruction whose input operand is initialized through parameter p of the function f. In this case, we determine the new slicing criterion as the parameter variable corresponding to the parameter p. To locate this parameter variable, we use *caller-callee relationship* and the *callee's function prototype*. In particular, we detect the call site for function f and analyze f's function prototype to obtain the index of the parameter corresponding to p. For example, the intraprocedural backward slice w.r.t. the `target_dllname` in Figure 4.1 uses the first parameter, *i.e.*, `pImgDelayDesc`, of `__delayLoadHelper2`. As two call sites on lines 5–7 and lines 14–16 invoke `__delayLoadHelper2`, we choose their first parameter variables, *i.e.*, `pDelayDesc1` on line 6 and `pDelayDesc2` on line 15, as the new slicing criterion.

Once the new slicing criterion is determined, we construct the interprocedural backward slice by composing the intraprocedural backward slices and use the composed slice in the emulation phase. One simple method for composing the intraprocedural slices is to collect the instructions of each intraprocedural backward slice. For example, the interprocedural backward slice w.r.t. the `target_dllname` in Figure 4.1 consists of the instructions of three intraprocedural backward slices w.r.t. the slicing criteria, *i.e.*, `target_dllname`, `pDelayDesc1`, and `pDelayDesc2`. However, this simple method produces *context-insensitive* slices, making the emulation phase complex. In particular, when emulating each instruction of the context-insensitive slice, we have to assume that the values of its operands are determined under all of its calling contexts.

To better support emulation, we combine the intraprocedural backward slices to construct a set of *context-sensitive interprocedural backward slices*. In particular, for a given intraprocedural backward slice s, if multiple new slicing criteria, $p_1 \dots p_n$, are determined, the set of the context-sensitive slices are constructed as $\{s_i \cup s | s_i = \cup_{p_i}$ intraprocedural backward slice w.r.t. p_i where $1 \leq i \leq n\}$. Thus, we can more straightforwardly use the context-sensitive slices to compute possible concrete values of the target component specification. For example, we can compute the possible values of `target_dllname` by emulating these slices in Figure 4.2. We describe more details of our backward slicing phase in Algorithm 2.

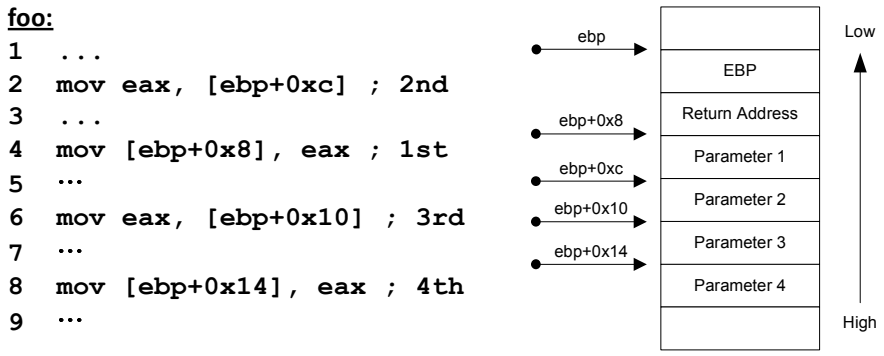
Algorithm 2 Backward Slicing Phase

Input: sc (a slicing criterion)
Output: Slices (a set of backward slices for sc)
Assumption: no recursion
Auxiliary functions:
Func(P): return the function that contains $p \in P$
UsedParms(f, s): return f's parameters used by s
CallSites(f): return call sites that invoke f
PredSubG(f, p): return a predecessor subgraph w.r.t. p over f's CFG
SubPDG(f, P): return the PDG based on $\cup_{p \in P}$PredSubG(f, p)
BSlice(P, pdg): return $\cup_{p \in P}$ an intra-backward slice w.r.t. p, which is computed by traversing pdg
UsedParmVars(cs, s): return parameter variables w.r.t. a call site cs that are used by a slice s

```
1:  Slices = {}
2:  WorkList = {({sc}, ∅)}
3:  while WorkList ≠ ∅ do
4:     Select and remove a (P, S) from WorkList
5:     f ← Func(P)
6:     g ← SubPDG(f,P)
7:     S' ← BSlice(P,g) ∪S
8:     if UsedParms(f,S')≠ ∅ then
9:        CS ← CallSites(f)
10:       for all cs ∈ CS do
11:          SC ← UsedParmVars(cs, S')
12:          Insert (SC, S') into WorkList
13:       end for
14:    else
15:       Insert S' into Slices
16:    end if
17: end while
```

Function Prototype Analysis. The backward slicing phase relies on function prototypes, but such information is often unavailable in binary code. Our solution to this problem is as follows. For a given function f, its parameters are stored in fixed locations during f's execution. Thus, we infer its prototype by analyzing how the instructions of the function access the memory chunks for the parameters, *i.e.*, read or write.

Figure 4.7 shows an example of our proposed prototype analysis for the `foo` function. Suppose that Figures 4.7(a) and 4.7(b) show part of `foo` and the stack layout at the beginning of the function's execution, respectively. In this case, the `idx`-th parameter is stored at the address $\texttt{ebp} + 4 \times (\texttt{idx} + 1)$ where the stack is aligned by four bytes. From this observation, we can infer



(a) Parameter access. (b) Stack layout.

Figure 4.7: Function prototype analysis.

`foo`'s prototype. It reads data from the memory chunks for its second and third parameters, and initializes the memory chunks for its first and fourth parameters, *i.e.*, its function prototype is `"eax foo(inout,in,in,inout)"`. Here we assume that its result is returned through the `eax` register.

To improve the precision of our prototype inference, we use the following effective heuristic. If the effective address of the memory chunk, obtained by the `lea` instruction, is passed to the function, we consider it as the `inout` parameter. The effective address corresponds to a pointer variable and the memory chunk that it points to is often initialized during function execution. Although this heuristics may increase the size of the computed slice, it is sufficient to compute possible values of the slicing criteria via emulation.

Emulation Phase. In this phase, we compute the possible values of the target component specification by emulating its corresponding context-sensitive slices. There are three challenges for slice emulation. The first challenge is how to schedule the instructions because we do not know their runtime execution sequence. If the instructions are incorrectly scheduled, they may violate the data and control flow dependencies among them, which may lead to imprecise results or emulation failures. The second challenge is how to pass function parameters. Although parameter passing captures useful data flow dependencies, the context-sensitive slices do not explicitly specify the dependencies. The third challenge is how to handle the call site instructions. Because we perform the data flow analysis by considering a call site as an instruction, the backward slice does not contain detailed code of the callee function.

Scheduling Algorithm. To develop a practical scheduling algorithm, we have analyzed all 682

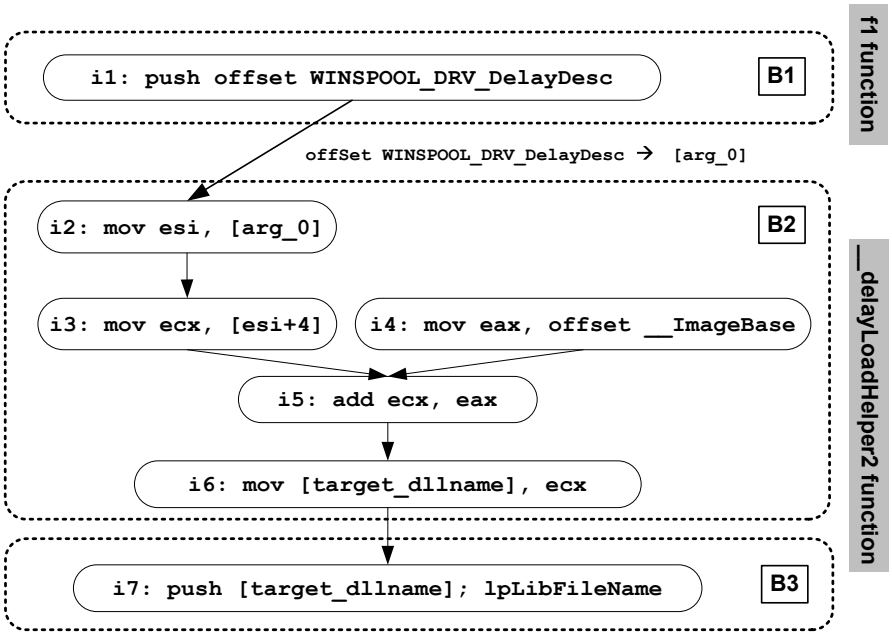


Figure 4.8: Data-flow dependency among basic blocks.

backward slices extracted from nine popular Windows applications (*cf.*, Table 4.1). We have observed that all the extracted slices form directed acyclic graphs. Therefore, we schedule the basic blocks in their topological order w.r.t. dataflow dependency. We then determine the order of the instructions of each basic block w.r.t. their sequence in the original code. For example, Figure 4.8 shows the data flow dependency among the basic blocks of the first slice in Figure 4.2. In this case, we schedule the basic blocks as `B1`, `B2`, and `B3`. For each basic block, the sequence of its instructions is determined as follows: `i1`, `i4,i2,i3,i5,i6`, and `i7`. The scheduled sequence of the instructions does not violate the data- and control-flow dependency among them.

Parameter Passing. To handle parameter passing, we initialize the stack frame before emulating the callee function. In particular, suppose that a parameter p is passed to a function f. In this case, before emulating f's basic blocks, we reserve the stack frame and initialize its memory chunk for the parameter with the concrete value of p. The location of the memory chunk is determined by the index of the passed parameter. For example, the address of the memory chunk for the `idx`-th parameter can be computed by $\texttt{ebp} + 4 \times (\texttt{idx} + 1)$, (*cf.* Figure 4.7).

For example, we handle the parameter passing from `f1` to `__delayLoadHelper2` in Figure 4.8. When `B1` is emulated, `offset WINSPOOL_DRV_DelayDesc` is stored on top of the call stack for `f1`. Assuming that the initial value of `esp` for emulating `B2` is equal to `0x13f258`, the stored value

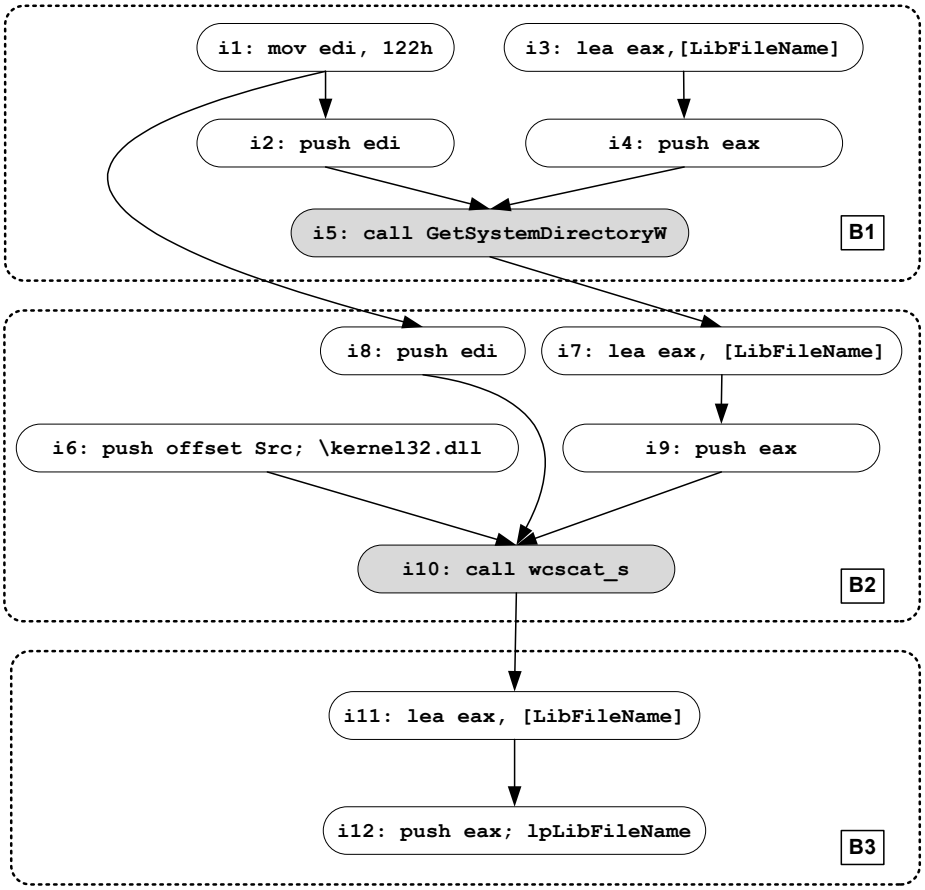
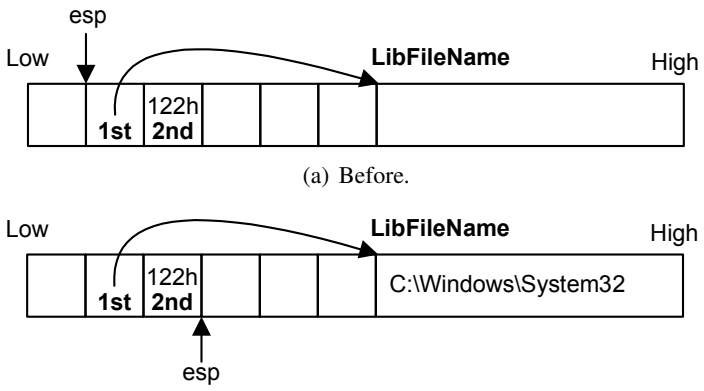


Figure 4.9: Backward slice with external library calls.

initializes a memory chunk at `arg_0=0x13f258+4×2`, because it corresponds to the first parameter to `__delayLoadHelper2`. The instructions use `arg_0` to reference the first parameter (*e.g.*, `i2`).

Call Site Instruction. To obtain the possible values of the target component specification, it is necessary to emulate the call site instruction. If the code of the invoked function resides in the current file, we can simply emulate the corresponding code. However, if the call site invokes a system call, we may not be able to obtain the code from the current file. Figure 4.9 shows an example slice with external library calls where each edge represents data flow dependency between two instructions. The slice determines the fullpath of the target component by concatenating the path to the system directory with a string `\kernel32.dll`. In this case, the instructions invoked by `i5` and `i10` are not available in the current file. In particular, `GetSystemDirectoryW` and `wcscat_s` are implemented in `KERNEL32.DLL` and `MSVCRT.DLL`, respectively.

One natural solution is to perform instruction-level emulation over the system call implementa-



(a) Before.

(b) After.

Figure 4.10: Side effects of `i5` in Figure 4.9.

tions obtained from the corresponding libraries. However, this is not practical because system call implementations typically have a large number of instructions and lead to poor scalability.

Thus, we do not emulate the system call code at the instruction-level. Instead, we use code to model the side effects of system calls and execute the models. For example, Figure 4.10(a) and Figure 4.10(b) show the stack layout before and after processing `i5` shown in Figure 4.9. The example models the side effect of `GetSystemDirectoryW`: 1) retrieve the two parameters from the stack; 2) obtain the system directory path by invoking `GetSystemDirectoryW`; 3) write the directory path to the memory chunk pointed to by the first parameter; 4) copy the system call's return value to `eax` register and adjust the `esp` register to clean up the stack frame.

Based on the technique discussed above, we can emulate the context-sensitive slices to compute the possible values of the target component specification. For example, we can compute the value, `"C:\Windows\System32\KERNEL32.DLL"`, of `lpLibFileName` by emulating the backward slice in Figure 4.9.

4.3.4 Checking Phase

In this phase, we evaluate the safety of the target component specifications obtained from the extraction phase. To this end, for each specification, we check whether or not the safety conditions in Definition 2.1.6 are satisfied. In particular, when the fullpath is specified, we check whether or not the specified file exists in the normal file system. For the filename specification, we consider that a specification can lead to unsafe loading if the target component is unknown and the

OS cannot resolve it in the directory that is first searched on the normal file system. Note that the names of the known component and the first directory searched by the OS for the resolution are predefined [43, 45, 94].

As an example of this phase, we check the component loading discussed in the attack scenario in Section 4.1. When opening the `.asx` file, `Winamp 5.58` tries to load `rapi.dll`. In this case, OS iterates through a list of predefined directories [45] to locate the file named `rapi.dll`. However, no such file is found during the iteration. Thus, this loading is unsafe, because attackers can hijack this loading by placing malicious `rapi.dll` files in the checked directories. In particular, the current working directory, one of the directories, is determined as the same directory as the `.asx` file, leading to the remote code execution attack. Suppose that the file named `rapi.dll` exists in the directory first searched, *i.e.*, the `Winamp` program directory. In this case, this loading is safe, because there is no directory such that attackers can misuse for hijacking.

4.4 Empirical Evaluation

In this section, we evaluate our static technique in terms of precision, scalability, and code coverage. We show that our technique scales to large real-world applications and is precise. It also has good coverage, substantially better than the proposed dynamic approach in Chapter 3.

4.4.1 Implementation

We implemented our technique on Windows XP SP3 as a plugin to IDA Pro [69], a state-of-the-art commercial binary disassembler. Our IDA Pro plugin is implemented using IDAPython [70] and three libraries: 1) NetworkX [116] for graph analysis, 2) PyEmu [121] for emulation, and 3) pefile [120] for PE format analysis.

For the precise analysis of binaries, it is important to map between C-like variables and memory regions accessed by instructions. We adapt the concept of a *abstract location* (*a-loc*) [13], which models a concrete memory address in terms of the base address for a memory region and a relative offset. For example, the a-loc for `&a[4]` is `mem_4` where `mem` is the base address of the array `a` and `4` is the relative offset from the base address. Refer to Balakrishnan and Reps [13] for more details.

Backward slicing phase in our technique uses function prototypes of system calls. To this end,

we analyzed the files in the system directory and collected prototypes for 3,291 system calls.

To emulate the code modeling side effects of system calls, we need to determine what system call is invoked through a given call site instruction. We have extended the `set_library_handler` function of PyEmu so that it can register callback functions for external function calls. We implemented the callbacks for 68 system calls used by the extracted slices.

To implement our tool, it is necessary to extract CFGs and call graphs from binaries. We leverage the disassemble result of IDA Pro in our current implementation. It is well-known that indirect jumps can be difficult to resolve for binaries. Although IDA Pro does resolve certain indirect jumps, it may miss control-flow and call dependencies, which is one source of incompleteness in our implementation.

4.4.2 Evaluation Setup and Results

We aim at detecting unsafe component loadings in applications. Because the detection of unsafe loadings from the system libraries is performed by the operating system, we only resolve the application components in the extraction phase.

The checking phase for a target specification requires the information on the first directory searched by the OS for the resolution and the relevant normal file system state (*cf.*, Definition 2.1.6 and Section 4.3.4). We obtain this information by analyzing the extracted parameters and the applications. For example, suppose that an application p loads an unknown component by invoking `LoadLibrary` with the component's filename. In this case, we can infer the directory where p is installed because Microsoft Windows first checks the directory where p is loaded. Regarding the normal file system state, we installed the applications with the default OS configuration and detected unsafe loadings for each application. In this setting, we assume that 1) the default file system state is normal, and 2) the installation of a benign application does not cause installed applications to have unintended component loadings.

Detection Results and Scalability

Table 4.1 and Table 4.2 show our analysis results on nine popular Windows applications. We chose these applications as our test subjects because they are important applications in wide-spread use.

	Resolved files			Context-sensitive Emulation				
	#	Size (MB)	Disasm. time	Call sites	Slices	Slice inst. (#) mean	Slice inst. (#) max	Failures
Acrobat Reader 9.3.2	18	38.2	34m 12s	85	145	5.1	40	34
Firefox 3.0	13	12.5	10m 48s	21	25	2.7	26	3
iTunes 9.0.3	2	25.1	11m 32s	53	128	13.7	187	74
Opera 10.50	3	11.6	12m 46s	28	30	3.0	29	2
Quicktime 7.6.5	17	40.5	9m 15s	70	119	13.5	54	58
Safari 5.31	24	37.5	11m 03s	72	137	5.8	48	33
Seamonkey 2.0.4	15	14.5	20m 44s	34	40	1.7	24	2
Thunderbird 3.0.4	15	15.0	19m 38s	34	40	1.7	24	2
Foxit Reader 3.0	2	10.2	5m 20s	18	18	2.1	13	5

Table 4.1: Analysis of the static detection.

	Loadtime	Runtime
Acrobat Reader 9.3.2	12 / 109	40 / 111
Firefox 3.0	9 / 77	12 / 22
iTunes 9.0.3	18 / 36	31 / 54
Opera 10.50	8 / 28	11 / 28
Quicktime 7.6.5	19 / 109	19 / 61
Safari 5.31	16 / 158	67 / 104
Seamonkey 2.0.4	9 / 88	20 / 38
Thunderbird 3.0.4	9 / 88	20 / 38
Foxit Reader 3.0	10 / 24	6 / 13

Table 4.2: Ratio of unsafe to total specifications.

The results show that our technique can effectively detect, from program binaries, unsafe component loadings potentially loaded at runtime. One interesting finding to note is that the results of the extraction phase for `Seamonkey` and `Thunderbird` are identical. This is likely because both applications are part of the Mozilla project and use the same set of program components.

We rely on IDA Pro for disassembling binaries, and Table 4.1 includes the time that it took IDA Pro to disassemble the nine applications. This time dominates our analysis time as we show later. These are large applications, and also we only need to disassemble the code once for all the subsequent analysis.

According to our analysis of context-sensitive emulation, the number of slices is generally larger than that of the call sites. This indicates that parameters for loading library calls can have multiple values, confirming the need for context-sensitive slices. The average number of instructions for the slices is quite small, which empirically validates our analysis design decisions.

Software	Open (s)	Call site (s)	Slicing (s)	Emulation (s)	Total (s)
Acrobat Reader 9.3.2	95.68	0.03	3.11	6.17	104.93
Firefox 3.0	41.69	0.03	0.19	0.22	42.13
iTunes 9.0.3	15.47	0.03	23.53	16.80	55.83
Opera 10.50	15.35	0.03	0.20	0.57	16.15
Quicktime 7.6.5	46.70	0.02	4.65	25.64	77.01
Safari 5.31	48.34	0.02	1.96	3.70	54.02
Seamonkey 2.0.4	37.51	0.02	0.19	0.52	38.24
Thunderbird 3.0.4	37.22	0.02	0.22	0.53	37.99
Foxit Reader 3.0	12.08	0.01	0.17	0.28	12.54

Table 4.3: Detection time.

Software	# of analyzed functions				# of inst. of analyzed functions			
	Demand-driven			Static	Demand-driven			Static
	mean	max	total	total	mean	max	total	total
Acrobat Reader 9.3.2	1.4	3	205	264,551	48.4	220	7,019	9,907,069
Firefox 3.0	1.0	1	25	63,550	34.4	158	859	3,071,548
iTunes 9.0.3	2.2	5	280	42,689	222.3	7,017	28,460	3,612,724
Opera 10.50	1.0	1	30	54,387	28.1	140	843	2,789,126
Quicktime 7.6.5	1.9	7	221	63,995	84.4	1,542	10,038	4,885,911
Safari 5.31	1.5	7	201	80,899	49.5	500	6,788	5,058,285
Seamonkey 2.0.4	1.0	1	40	79,636	30.9	125	1,236	3,840,465
Thunderbird 3.0.4	1.0	1	40	78,520	30.9	125	1,236	3,782,799
Foxit Reader 3.0	1.2	3	22	56,439	22.8	72	411	2,032,545

Table 4.4: Relative cost of slice construction.

We now discuss the evaluation of our tool's scalability. To this end, we measure its detection time and the efficiency of its backward slicing phase. Table 4.3 shows the detailed results of detection time. The results show that our analysis is practical and can analyze all nine large applications within minutes. To further understand its efficiency, we compared cost of our backward slicing with one of standard SDG-based slicing. Although we do expect to explore fewer instructions with a demand-driven approach, we include the comparison in Table 4.4 to provide concrete, quantitative data. For a standard SDG-based approach, one has to construct the complete SDG before performing slicing. We thus measured how many functions and instructions there are in each application as these numbers indicate the cost of this *a priori* construction (*cf.* the two columns labeled "Static total"). As the table shows, we achieve orders of magnitude reduction in terms of both the number of functions and the number of instructions analyzed.

Software	Component loadings			Unsafe loadings			Static reachability	
	Dynamic	Static	∩	Dynamic	Static	∩	Reachable	Unknown
Acrobat Reader 9.3.2	14	111	11	2	40	1	32	7
Firefox 3.0	16	22	11	6	12	4	1	7
iTunes 9.0.3	5	54	2	3	31	1	29	1
Opera 10.50	20	28	13	9	11	4	7	0
Quicktime 7.6.5	6	61	4	2	19	1	9	9
Safari 5.31	27	104	24	17	67	15	52	0
Seamonkey 2.0.4	24	38	12	9	20	6	0	14
Thunderbird 3.0.4	25	38	11	6	20	5	0	15
Foxit Reader 3.0	6	13	1	0	6	0	6	0

Table 4.5: Static detection versus dynamic detection in Chapter 3.

Comparison with Dynamic Detection

To evaluate our tool's code coverage, we compare unsafe loadings detected by the static and dynamic analyses. In particular, we detected unsafe component loadings with the existing dynamic technique [94] and compared its results with our static detection. To collect the runtime traces, we executed our test subjects one by one with relevant inputs (*e.g.*, PDF files for Acrobat Reader) and collected a single trace per application. Please note that the dynamically detected unsafe loadings are only a subset of all real unsafe loadings.

In this evaluation, we focus on application-level runtime unsafe loadings as loadtime dependent components are loaded by OS-level code. Table 4.5 shows the detailed results. We see that our static analysis can detect not only most of the dynamically-detected unsafe loadings but also many additional (potential) unsafe loadings, most of which we believe are real and should be fixed. Next we closely examine the results.

Static-only Cases.

Our static analysis detects many additional potential unsafe loadings. We carefully studied these additional unsafe loadings manually. In particular, we analyzed whether they are reachable from the entry points of the programs, *i.e.*, whether there exist paths from the entry points to the call sites of the unsafe loadings in the programs' interprocedural CFGss (ICFGs). In this analysis, we consider the main function of an application and the UI callback functions as the entry points of the application's ICFG. Table 4.5 shows our results on this reachability analysis. Note that those loadings marked as "Unknown" may still be reachable as it is difficult to resolve indirect jumps

in binary code, so certain control flow edges may be missing from the ICFGs. All the statically reachable unsafe loadings lead to component-load hijacking if 1) the corresponding call sites are invoked and 2) the target components have not been loaded yet.

Although it is difficult to trigger the detected call sites dynamically (due to the size and complexity of the test subjects), we believe most of the call sites are *dynamically reachable* as dead-code is uncommon in production software. As a concrete example of unsafe loading, Foxit Reader 3.0 has a call site for loading `MAPI32.DLL`, which is invoked when the current PDF file is attached to an email message. This loading can be hijacked by placing a file with the same name `MAPI32.DLL` into the directory where Foxit Reader 3.0 is installed.

Dynamic-only Cases. According to Table 4.5, our technique misses a few of the dynamically detected unsafe loadings. We manually examined all these cases, and there are two reasons for this: *system hook dependency* and *failed emulation*, which we elaborate next.

First, Microsoft Windows provides a mechanism to hook particular events (*e.g.*, mouse events). If hooking is used, a component can be loaded into the process to handle the hooked event. This component injection introduces a system hook dependency [4]. Such a loading may be unsafe, but since it is performed by the OS at runtime and is not an application error, we do not detect it.

Second, our extraction phase may miss some target component specifications due to failed emulations. If this happens, we may miss some unsafe loadings even if their corresponding call sites are found. Emulation failures can be caused by the following reasons.

External Parameters. A target specification may be defined by a parameter of an exported function, which is not invoked. For example, suppose that a function `foo` exported by a component `A` loads a DLL specified by `foo`'s parameter. If `foo` is not invoked by `A`, the parameter's concrete value will be unknown. One may mitigate this issue by analyzing the data flow dependencies among the dependent components. However, such an analysis does not guarantee to obtain all the target specifications, because the exported functions are often not invoked by the dependent components.

Uninitialized Memory Variables. The slices may have instructions referencing memory variables initialized at runtime. In this case, our slice emulation may be imprecise or fail. To address this problem, it is necessary to extract the sequence of instructions from the dependent components that initialize these memory variables and emulate the instructions before slice emulation. Although it

is possible to analyze memory values, such as the Value Set Analysis (VSA) [123], it is difficult to scale such analysis to large applications.

Imprecise Inferred Function Prototypes. Our technique infers function prototypes by analyzing parameters passed via the stack. However, function parameters may be passed via other means such as registers. For example, the `__fastcall` convention uses `ECX` and `EDX` to pass the first two parameters. Therefore, when function parameters are passed through unsupported calling conventions, the inferred function prototypes may omit parameters that determine the new slicing criteria. For example, suppose that we extract a context-sensitive sub-slice s from a function `foo`, and `ECX` is used as a parameter variable of s. In this case, we do not continue the backward slicing phase, because the inferred prototype does not contain `ECX`. Although imprecisely inferred function prototypes may lead to emulation failure, our results show that this rarely happens in practice—we observed only 14 cases out of a total of 213.

Unknown Semantics of System Calls. Detailed semantics of system calls is often undocumented, and sometimes even their names are not revealed. When we encounter such system calls, we cannot analyze nor emulate them. When information of such system calls becomes available, we can easily add analysis support for them.

Disassemble Errors. Our implementation relies on IDA Pro to disassemble binaries, and sometimes the disassemble results are incorrect. For example, IDA Pro sometimes is not able to disassemble instructions passing parameters to call sites for delayed loading. Such errors can lead to imprecise slices and emulation failures.

4.5 Related Work

We survey additional related work besides the one on dynamic detection of unsafe loadings [94], which we have already discussed.

Our technique performs static analysis of binaries. Compared to the analysis of source code, much less work exists [5,13,14,35,37,90,92,123,137]. In this setting, Value Set Analysis (VSA) [13, 123] is perhaps the most closely related to ours. It combines numeric and pointer analyses to compute an over-approximation of numerical values of program variables. Compared to VSA, our

technique focuses on the computation of string variables. It is also demand-driven and uses context-sensitive emulation to scale to real-world large applications.

Starting with Weiser's seminal work [157], program slicing has been extensively studied [143, 162]. Our work is related to the large body of work on static slicing, in particular the SDG-based interprocedural techniques. Standard SDG-based static slicing techniques [5, 23, 66, 119, 124, 134] build the complete SDGs beforehand. In contrast, we build control- and data-flow dependence information in a demand-driven manner, starting from the given slicing criteria. Our slicing technique is also modular because we model each call site using its callee's inferred summary that abstracts away the internal dependencies of the callee. In particular, we treat a call as a non-branching instruction and approximate its dependencies with the callee's summary information. This optimization allows us to abstract away detailed data flow dependencies of a function using its corresponding call instruction. We make an effective trade-off between precision and scalability. As shown by our evaluation results, function prototype information can be efficiently computed and yield precise results for our setting.

Our slicing algorithm is demand-driven, and is thus also related to demand-driven dataflow analyses [67, 122], which have been proposed to improve analysis performance when complete dataflow facts are not needed. These approaches are similar to ours in that they also leverage caller-callee relationship to rule out infeasible dataflow paths. The main difference is that we use a simple prototype analysis to construct concise function summaries instead of directly traversing the functions' intraprocedural dependence graphs, *i.e.*, their PDGs. Another difference is that we generate context-sensitive executable program slices for emulation to avoid the difficulty in reasoning about system calls.

As we discussed earlier, instead of emulation, symbolic analysis [85, 130] could be used to compute concrete values of the program variables. However, symbolic techniques generally suffer from poor scalability, and more importantly, it is not practical to symbolically reason about system calls, which are often very complex. The missing implementation for undocumented system calls is the challenge for emulation, while for symbolic analysis, complex system call implementation is an additional challenge. We introduce the combination of slicing and emulation to address this additional challenge. Our novel use of context-sensitive emulation provides a practical solution for computing the values of program variables.

4.6 Conclusion and Future Work

We have presented a practical static binary analysis to detect unsafe loadings. The core of our analysis is a technique to precisely and scalably extract which components are loaded at a particular loading call site. We have introduced context-sensitive emulation, which combines incremental and modular slice construction with the emulation of context-sensitive slices. Our evaluation on nine popular Windows application demonstrates the effectiveness of our technique. Because of its good scalability, precision, and coverage, our technique serves as an effective complement to dynamic detection [94]. For future work, we would like to consider two interesting directions. First, because unsafe loading is a general concern and also relevant for other operating systems, we plan to extend our technique and analyze unsafe component loadings on Unix-like systems. Second, we plan to investigate how our technique can be improved to reduce emulation failures.

Chapter 5

Automatic Detection of Insecure Component Usage

5.1 Introduction

Component-based development has been a major paradigm for engineering software. In particular, a client application can perform desired functionalities by invoking interface calls of a component. This paradigm allows better code reuse and makes software development more productive. For example, Trident [145], a browser layout engine developed by Microsoft, has been used in IE and many other Windows applications.

Although component reuse has significant benefits, it may lead to security vulnerabilities if a component is not used properly in its client software. The following example, which we first discovered through a manual examination, inspired this research. IE 9 enables an XSS filter by default [71]. However, IE-based browsers, such as IE Tab, use the same browser components as IE, but do not enable the XSS filter. This insecure component usage makes these IE-based browsers vulnerable to XSS attacks. As this example shows, insecure component usage can cause serious vulnerabilities in component-based software. However, this problem has not been much explored. Previous work on component security has focused on designing and developing frameworks for secure component usage [18,61,62,88,139,153], detection of insecure components [15,41,63,118], and surveys on component security issues [40,58].

In this chapter, we present a *differential analysis framework* [104] to detect and analyze insecure component usage in component-based software. Here is the key idea behind our framework. Suppose that two applications, a reference `A` (which we assume to be correct and secure w.r.t. component usage) and a test subject `B`, reuse components that check security policies to block malicious activities. If `A` and `B` configure or evaluate the policies inconsistently, `B` may have unprotected runtime execution. In the XSS filter example earlier, IE acts as the reference, and IE Tab uses `URLMON.dll` insecurely because it neither configures the built-in security policies for XSS filter nor utilizes them to block XSS attacks.

To realize our framework, there are two main technical challenges: 1) how to extract the configurations of security policies maintained by a component, and 2) how to detect potential insecure component usage of a client software.

Extracting Policy Configurations. To extract a policy configuration, we monitor the writes to component memory space that potentially stores security policy configurations. The memory writes provide us with the following important information: 1) instructions that configure relevant security policies, 2) locations of the buffers to store the policies, and 3) concrete configuration data. For example, `URLMON.dll` maintains a memory buffer in its global data region to store the URL action policies. IE configures the policies via memory writes.

To check a security policy, an application retrieves its configuration data from the relevant memory buffer and uses the data for comparison. In this case, if a reference and a test subject configure the same security policy in an inconsistent manner, the comparison results are different, making them take different execution paths. Based on this idea, we define missing and incorrect configurations that can lead to insecure component usage.

A *missing configuration* corresponds to the case where the reference only configures and checks a particular set of security policies. Thus, the test subject is vulnerable to attacks that can be blocked by these policies. The XSS filter example belongs to this category. An *incorrect configuration* corresponds to the case where both the reference and the test subject configure and check a particular set of security policies but their different configuration data cause inconsistent subsequent execution paths. For example, while IE enables `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE`, IE Tab does not. The configuration of this security policy is checked by both IE and IE Tab at runtime, but

the inconsistent configuration data lead them to behave differently. Specifically, IE Tab allows user names and passwords in a URL address, leading to potential attack vectors for phishing [79]. We provide a detailed analysis of this issue in Section 5.2.

Detecting Inconsistent Policy Configurations. As we discussed earlier, the inconsistent configuration of a security policy leads to inconsistent subsequent execution patterns. For detection, we capture control flows triggered by the configuration data from the reference and the test subject at runtime and compare them. To capture control flow information, we determine conditional branches whose evaluations are potentially affected by the configuration data via static binary analysis and capture information regarding whether or not each conditional branch has been taken at runtime.

From these observations, we design our differential analysis framework as a three-phase analysis: (**P1**) *detecting potential policy evaluation*, (**P2**) *extracting policy-related execution*, and (**P3**) *detecting inconsistent policy configurations*.

P1: ***Detecting Potential Policy Evaluation.*** This phase detects information related to policy evaluation from dynamic execution of the reference and static properties of a target component. To this end, we detect instructions that read data from component memory space at runtime. Afterward, we perform static forward data slicing to detect conditional jumps that can be affected by the data. If such conditional jumps exist, the data can control the subsequent execution paths at runtime. Thus, the detected instructions potentially read the configuration data and evaluate relevant security policies. We use this information to perform subsequent analyses scalably.

P2: ***Extracting Security Policy-related Execution.*** This phase extracts software execution related to the policy configuration and evaluation performed by the reference and the test subject. To capture the policy configuration, we detect memory writes to component memory space at runtime. Regarding policy evaluation, we log the memory reads and the comparison results on the conditional jumps detected in the previous phase.

P3: ***Detecting Inconsistent Policy Configurations.*** This phase analyzes inconsistency of policy-controlled executions between the reference and the test subject to detect missing and incorrect configurations. In particular, we determine whether or not the conditional jumps relevant to a particular security policy are evaluated consistently.

For evaluation, we implemented our framework for Windows applications and applied it to detect inconsistent policy configurations in reusing popular software components. Our results show that inconsistent policy configurations happen frequently and lead to security vulnerabilities. In particular, we detected several insecure usages of the browser components that disable default protection mechanisms of IE 9. Our framework can also precisely locate root causes of the detected insecure usages, which can help developers fix any detected vulnerabilities and securely reuse software components. The results also show that our framework is scalable. For example, it took less than 15 minutes total to detect inconsistent policy configurations in reusing `URLMON.dll` across all six analyzed browsers.

This chapter makes the following main contributions:

- We introduce and formalize insecure component usage in terms of inconsistent configurations and evaluations of security policies.
- We develop the first practical framework based on differential analysis to detect inconsistent policy configurations. Our framework works directly on software binaries; source code is not needed.
- We implement our framework as a practical tool and evaluate its effectiveness by detecting and analyzing insecure usage of widely-used components in real-world software.

The remainder of this chapter is structured as follows. We describe our differential analysis framework for detecting inconsistent policy configurations in Section 5.2. Section 5.3 discusses implementation details of our framework for Microsoft Windows applications. We then evaluate effectiveness of our framework by using it to detect and analyze insecure component usage (Section 5.4). Finally we survey related work (Section 5.5) and conclude (Section 5.6).

5.2 Detection Framework

In this section, we present a framework to detect inconsistent policy configurations defined in Section 2.2.

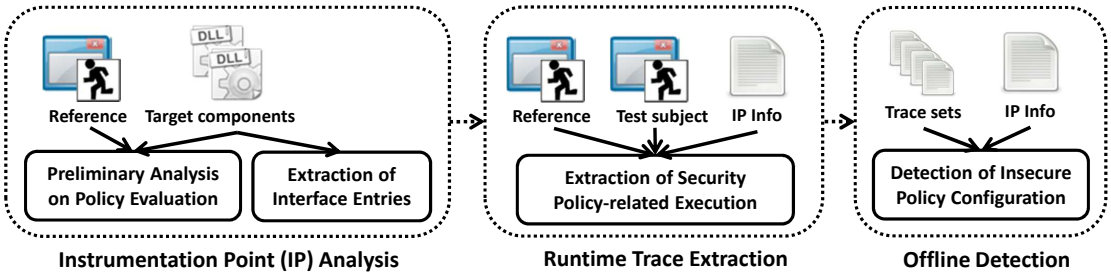


Figure 5.1: Detection framework.

5.2.1 Overview

To detect inconsistent policy configurations formalized in Definitions 2.2.6 and 2.2.7, it is necessary to analyze how security policies are set and enforced in code. However, it is challenging, because 1) most components are distributed as binaries, and 2) it is difficult to know which memory locations are used for security policy configuration and evaluation. To address this issue, at the high-level, we design our framework as two phases: *runtime extraction* and *offline detection*. In the runtime extraction phase, we instrument runtime executions of a reference and a test subject to capture security policy-related executions (Definition 2.2.4). We perform an offline analysis to detect insecure component usage in the captured executions.

Although the high-level approach appears straightforward, the main challenge is how to perform scalable and precise analysis. For example, IE performs millions of memory accesses at runtime, and it is practically infeasible to instrument and analyze all of them.

To address this scalability issue, our framework uses the following optimizations: 1) instrumenting target component execution, 2) filtering irrelevant memory accesses, and 3) performing preliminary analysis on policy evaluation.

Instrumenting Target Component Execution. Instrumenting all instructions executed by an application at runtime suffers from poor scalability. To mitigate this issue, our framework only instruments components of interest at runtime, because the configuration state maintained by the components is generally accessed by their code. Suppose that the configuration state M is maintained by a component A. When other components access M, they generally invoke relevant interface calls of A that access M.

It is possible that other components can directly access M. However, this cannot be used in

component-based software development because the location of M cannot be reliably resolved. For example, the base addresses of the components loaded at runtime often change [132, 159], making locations of their global data regions inconsistent.

Filtering Irrelevant Memory Accesses. As we discussed, capturing all accesses to arbitrary memory space is not feasible. To mitigate this issue, we filter the memory accesses that are unlikely to be relevant to security policy-related executions.

Our *key insight* is that the code executed by any thread can access the configuration state maintained by a component at runtime. Suppose that two different threads configure and evaluate a particular security policy, respectively. In this case, both threads should access the same memory location. Otherwise, integrity issues on the configuration data would arise. Based on this observation, we filter the logging of accesses to thread-specific memory space such as the stack.

Performing Preliminary Analysis on Policy Evaluation. According to Figure 2.2, policy evaluations are conducted by 1) reading data from the configuration state and 2) determining whether or not the data is matched with a specified operand. In order to detect inconsistent policy configurations (Section 2.2.2), it is necessary to capture the results of policy evaluations at runtime. However, it again suffers from scalability problems, because the policies are generally evaluated by conditional jumps such as `JNE`, which are executed frequently to determine program flow at runtime. Thus, instrumenting all conditional jumps at runtime is not feasible in practice.

To address this problem, we perform a preliminary analysis to detect those conditional jumps affected by data reads from the configuration state. Our observation is that the evaluation of the conditional jumps are affected by the data reads from the configuration state (see Figure 2.2). We detect these conditional jumps via dynamic and static binary analyses. Specifically, we dynamically capture the memory reads from the configuration state under a given workload. Then we extract the conditional jumps potentially affected by them via static binary analysis. We use information on the memory reads and the conditional jumps to reduce logging of policy evaluations in the subsequent phase.

Based on these optimizations, we present our framework in Figure 5.1. The following sections illustrate details of each phase in our framework using a running example. In particular, we detect insecure component usage to allow potentially malicious URL addresses in IE-based browsers.

```
...
i1   cmp  _GlobalProcessDisableUserPswdForHttp, 0
i2   jz   i6
i3   xor  eax, eax
i4   inc  eax
i5   call GetUrlAddress(*,*,*,*,*,*,*,*,*,eax,*,*)
...
i6   xor  eax, eax
i7   jmp  i5
...
```

(a) Evaluation.

```
...
i1  push offset g_FEATURE_HTTP_USERNAME_PASSWORD_DISABLE
i2  call _CoInternetIsFeatureEnabledInternal
i3  dec  eax
i4  neg  eax
i5  sbb  eax, eax
i6  neg  eax
i7  mov  _GlobalProcessDisableUserPswdForHttp, eax
...
```

(b) Configuration.

Figure 5.2: Policy-related code example.

In the example, we access `http://www.microsoft.com` as the workload for detection and use `WININET.dll` as a target component.

5.2.2 Instrumentation Point Analysis

Preliminary Analysis on Policy Evaluation. As discussed, instrumenting all memory reads and conditional jumps is not feasible. To address this problem, we detect them in advance and instrument their executions in the subsequent analysis. Our key observation is that there exists data flow between the memory reads and the conditional jumps. According to Figure 2.2, the policy evaluation is affected by the configuration data read. Based on this, we locate the instructions relevant to the policy evaluation via dynamic and static program analyses. In particular, we 1) dynamically instrument the execution of target components to detect reading data from the configuration state and 2) perform static forward data slicing w.r.t. the detected memory reads to locate relevant conditional jumps. Note that we consider that heap or global data regions of target components can contain the configuration state, because they are not thread-specific (Section 5.2.1).

Figure 5.2(a) shows an example of policy-controlled execution by `WININET.dll`. Specifically,

`i1` reads the configuration data `_GlobalProcessDisableUserPswdForHttp` stored in the global data region of `WININET.dll` and determines whether or not the data is equal to zero. The evaluation result affects the invocation of the `GetUrlAddress` function at `i5` by deciding one of its parameters, *i.e.*, the value of `eax`.

To detect the instructions relevant to policy evaluation, we capture the instructions to read data from the global data region in `WININET.dll` during the workload via dynamic binary instrumentation (*i.e.*, `i1`). Then we extract the static forward slice w.r.t. the data read by `i1`. In this case, the slice contains `i1` and `i2`, because the `cmp` instruction at `i1` sets `ZF` according to the comparison result, and the `jz` instruction takes `ZF` to determine the next instruction to execute.

Once the slices w.r.t. the detected memory reads are extracted, we analyze them to locate the instructions relevant to the policy evaluation. In particular, suppose that S is a forward data slice w.r.t. a memory read instruction I. In this case, we consider I relevant to the policy evaluation only if S contains conditional jumps. This is because policy evaluations are generally performed by both memory reads and relevant conditional jumps (see Figures 2.2 and 5.2(a)). Based on this idea, we determine the memory reads and the conditional jumps in its forward slice as the instrumentation points in the subsequent analysis.

Note that we only perform this preliminary analysis for the reference software. This is because our framework does not detect security policies configured only in the test subject (Definitions 2.2.6 and 2.2.7). To detect them, we swap the test subject with the reference and repeat the analysis.

Extracting Interface Entries. As we have discussed, when a component A maintains the configuration state M, other components generally access M by invoking interface calls exported by A. Thus, information on the invoked interface calls provides us with detailed insight on insecure component usage.

To capture the information, our framework dynamically instruments the entry points of the interface calls exported by the target components. To this end, it is necessary to determine the interface entries as the instrumentation points. However, it is difficult to locate them at runtime, because the instructions at the entries are also frequently executed by non-entry code at runtime. For example, while the `push` instruction is often executed at the entry point as part of a function prologue, it is also used for parameter passing. To address this problem, we perform static binary analysis to

extract the interface call entries and pass them into the subsequent analysis as the instrumentation points.

Our key observation is that the addresses of the interface call entries are generally stored in data tables that can be read from outside of the component. Because components are developed as position-independent code, they generally support memory chunks that can be accessed by other components to resolve the virtual addresses of desired interface calls at runtime. For example, the PE [106] and ELF [49] formats have Export Table and Procedure Linkage Table to provide dynamic linking, respectively. We statically analyze data reference to the entries of all functions in the target components. We consider the function entries that have such data references as the instrumentation points to capture interface call invocations.

5.2.3 Runtime Trace Extraction

This phase extracts detailed information on security policy-related executions of the target components during the workload run by both the reference and the test subject.

In particular, we instrument the runtime execution of the target components to record the following: the policy configurations, their evaluations, and the invocations of the interface calls to the target components. We store the captured information to files for use in our offline analysis.

Policy Configuration. According to Definition 2.2.2, an application specifies security policies via memory writes to its configuration state. To capture policy configurations, we instrument the runtime information of the target components that perform data writes to non-thread specific memory regions (see Section 5.2.1). During instrumentation, we log the following information: addresses of the memory writes, values of the data written, and addresses of the memory written.

Figure 5.2(b) shows a policy configuration example by `WININET.dll` with the given workload. In particular, the code operates as follows. First, `i1-i6` initialize `eax` by determining whether `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE` is enabled. Next `i7` writes the value of `eax` to a memory buffer `_GlobalProcessDisableUserPswdForHttp` in the global data region of `WININET.dll`. To capture the security policy configuration, our framework instruments the execution of `i7` and logs the following information: address of `i7`, value of the `eax`, and address of `_GlobalProcessDisableUserPswdForH`

Policy Evaluation. To capture information on policy evaluations, our framework only instruments

executions of those instructions detected by the preliminary analysis from the previous phase. Note that this allows us to significantly reduce the instrumentation points for capturing policy evaluations (see Section 5.2.1).

Regarding the information to be captured, consider the following code execution for evaluating a policy: an instruction I reads a data D from a non-thread specific memory region Mem, and D determines whether a conditional jump $Cond$ is taken or falls through. In this case, our framework logs 1) the address of I, 2) the value of D, 3) the address of Mem, 4) the address of $Cond$, and 5) the evaluation result of $Cond$ (*i.e.*, taken or fall-through). For example, when instrumenting Figure 5.2(a), we capture the address of `i1`, the value of `_GlobalProcessDisableUserPswdForHttp`, the address of the memory read, and the evaluation result of `i1`.

Interface Call Entries. We capture invocations of the interface calls to the target components by instrumenting their statically-detected entry points. However, it is possible for a component to invoke its interface call at runtime. To detect this, we analyze the return addresses of invoked interface calls, which are stored on the top of the stack. In particular, suppose that we instrument an interface call entry f of a component C. In this case, we log f only if the return address of f is not part of the memory space corresponding to C. Based on this approach, we can precisely detect invocations of the interface calls to target components at runtime.

5.2.4 Offline Detection

From the previous analyses, we obtain the execution traces of the target components by the reference and the test subject under the given workload. Each trace contains a sequence of detailed information for each software with the following runtime information: 1) the policy configurations, 2) the policy evaluations, and 3) the invocations of the interface calls on the target components. Our offline phase detects inconsistent policy configurations from the traces as follows:

Extracting Policy Configurations and Their Evaluations. For each trace, we extract information on the configuration and the evaluation for each security policy. To this end, we first track the memory access patterns for each captured data address. For example, `i7` in Figure 5.2(b) writes a configuration data to `_GlobalProcessDisableUserPswdForHttp`, and `i1` in Figure 5.2(a) reads the data. Based on this memory access pattern, we can infer the instructions for configuring security

policies and reading their configuration data.

Once the instructions that read the configuration data are located, we can extract the evaluation results of their relevant conditional jumps. In particular, we sequentially search for the relevant conditional jumps, starting from the instructions, until the next read access of the configuration data is found. Next we retrieve their evaluation results captured during the instrumentation. Note that the result of the preliminary analysis provides us with the conditional jumps relevant to the instructions. For example, because `i1` affects the evaluation of `i2` (see Section 5.2.2), `i2` can be located by checking the instructions that follow `i1` in the trace.

Detecting Inconsistent Policy Configurations. The previous analysis step provides us with the policy configurations and their evaluation results for the reference and the test subject. We use this information to detect inconsistent policy configurations formalized in Definitions 2.2.6 and 2.2.7. Missing configurations are detected by finding those policy configurations and the associated evaluations that are only present in the reference, and incorrect configurations are detected by finding inconsistent evaluations of the relevant conditional jumps. In particular, a security policy is configured incorrectly in the test subject if the following conditions are satisfied: 1) both the reference and the test subject configure the same security policy and read its configuration data; 2) the policy evaluation results on the data are different. For example, while IE Tab takes a jump at `i2` in Figure 5.2(a), IE just continues the execution. This inconsistency shows that IE Tab does not enable `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE` whose configuration is stored in `WININET.dll`. This incorrect configuration makes IE Tab allow user names and password in URL address, which is blocked by IE [79]. Thus, IE Tab misuses `WININET.dll` w.r.t. the security policy `FEATURE_HTTP_USERNAME_PASSWORD_DISABLE`, making it vulnerable to phishing attacks [79].

Helping Developers to Securely Use Components. Our framework can extract interface calls for policy configurations and their evaluations. Because we capture all invocations of the interface calls to the target components, we can infer which interface calls perform policy configurations or their evaluations. For example, IE Tab invokes the `InternetSetOptionA` function to execute `i7` in Figure 5.2(b) for policy configuration.

This interface-level information can help developers to use components securely. For example, IE configures its policy by invoking the `InternetQueryOptionW` function, leading to its

correct policy configuration. Although both IE and IE Tab evaluate the policy while invoking `HttpSendRequestW`, the interface calls that configure the policy are different. This information can guide developers of IE Tab to securely use `WININET.dll`.

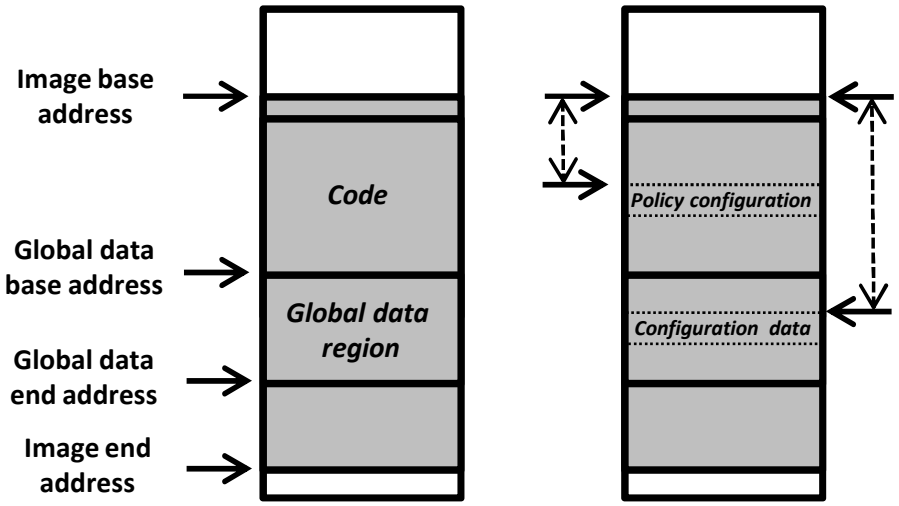
5.3 Implementation

To evaluate our technique, we have implemented it for Microsoft Windows applications. This section presents details on our implementation.

Overview. According to Figure 5.1, our framework consists of three phases: 1) instrumentation point analysis, 2) runtime trace extraction, and 3) offline detection. The main components for the first two phases are dynamic binary instrumentation and static binary analysis. To this end, we use Pin [100] for runtime instrumentation and have developed plugins for IDA Pro [69] (a state-of-the-art commercial binary disassembler) by using IDAPython [70] for binary analysis. When instrumenting each policy-related execution, we record not only the information of the captured instructions but also process and thread identifiers at runtime. This makes our offline analysis independent of thread interleavings. For the offline phase, we have developed Python scripts to analyze the traces obtained from the earlier phases.

Runtime Trace. As we discussed in Section 5.2.1, an application performs many memory accesses and conditional jumps. Although we filter those irrelevant ones, the number of captured instructions is still large. For example, when IE accesses the Google web page, we dynamically captured a large number (332,756) of memory accesses and conditional jumps. To store this large amount of information efficiently, we dump the captured information as binary data. As an example, we designed a data structure to store conditional jumps that contains the following information: an identifier for the conditional jump, process and thread identifiers, the address of the conditional jump, and its evaluation result. When capturing this information on a conditional jump, we fill the data structure and stores it as binary data of twenty bytes. Using this optimization, we can effectively analyze the large captured information in practice.

Instrumenting Component Code Execution. Our framework only instruments executions of target components at runtime. To this end, we dynamically instrument the loading of each image and



(a) Base and end addresses. (b) Relative offset.

Figure 5.3: Memory address space of loaded target component.

determine whether or not the image is one of the target components. If so, we use the base and end addresses of the image (see Figure 5.3(a)) to determine the instructions to instrument. In particular, we instrument executions of those instructions whose virtual addresses are part of the memory spaces of the target components. Note that a target component is loaded before its instructions are executed.

Filtering Irrelevant Memory Accesses. To capture policy configurations and evaluations, we focus on memory accesses to the global data region of a target component. To this end, when the target component is loaded, we extract the base and end addresses of its global data region (see Figure 5.3(a)). We use this information to detect those instructions that access the global data region.

Logging Virtual Address Information. Our framework captures information on virtual addresses for use by our offline detection. For example, we log information on the virtual addresses that read or write the configuration data. However, we cannot use the virtual address alone for offline detection because the same virtual address may not refer to the same location of interest in the target component. For example, suppose that a component C is loaded by the reference and the test subject at the memory spaces starting with different base addresses [132, 159]. In this case, the same virtual address in the offline phase does not refer to the same instructions or memory buffers.

To address this issue, when capturing a virtual address at runtime, we compute its relative offset

from the base address of the loaded target component (*e.g.*, Figure 5.3(b)) and log it. Using this approach, we can precisely extract the virtual address information independent of the base address of the loaded target component.

5.4 Empirical Evaluation

In this section, we evaluate how effective our technique is for detecting and analyzing insecure component usage in popular Windows applications. We show that 1) our framework can automatically detect inconsistent policy configurations in real-world software, and inconsistent policy configurations are prevalent and constitute a general security and reliability issue (Section 5.4.1); 2) our in-depth study of selected inconsistencies reveal new, serious vulnerabilities in widely-used software (Section 5.4.2); and 3) our framework can be effectively used for root-cause analyses to understand the detected inconsistent policy configurations and vulnerabilities (Section 5.4.3).

5.4.1 Prevalence of Inconsistent Policy Configurations

To evaluate our framework, we have first applied it to detect inconsistent policy configurations in real-world software. In particular, we have analyzed applications using widely-used components (such as the IE browser components and the Flash Player) and evaluated how our chosen reference programs and test subjects differ in terms of policy configurations under various workloads. Table 5.1 gives the detailed information, in particular how many detected inconsistent policy configurations in the test subjects w.r.t. their respective reference programs and workloads. Our results show that inconsistent policy configurations frequently occur in real-world, widely used applications. Note that all the reported inconsistencies are *real* and *detected fully automatically* by our tool by capturing and comparing inconsistent security policy evaluation patterns.

According to Definition 2.2.9, inconsistent policy configurations can lead to insecure component usage. Our framework automatically detects inconsistent policy configurations. A detailed analysis is needed to understand how security relevant they are. We perform such a detailed analysis of insecure component usages of major IE components [145] (*i.e.*, `MSHTML.dll`, `URLMON.dll`, and `WININET.dll`) in real-world IE-based browsers. We consider IE as a reference and the following browsers as test subjects: IE Tab 2 [72], Lunascape 6 [101], Slim Browser 5.01 [135],

Reference	Workload	Component	Test subject	Inconsistent policy configuration		Total
				Missing	Incorrect	
Internet Explorer 9	connect to microsoft.com	URLMON.dll	IE Tab 2	176	28	204
			Lunascape 6	167	33	200
			Slim 5.01	197	36	233
			Green 5.8	188	26	214
			WebbIE 3.14	175	27	202
			Enigma	190	25	215
		WININET.dll	IE Tab 2	215	18	233
			Lunascape 6	272	21	293
			Slim 5.01	229	19	248
			Green 5.8	235	14	249
			WebbIE 3.14	217	11	228
			Enigma	187	20	207
	login to gmail.com	URLMON.dll	IE Tab 2	43	12	55
			Lunascape 6	81	10	91
			Slim 5.01	45	16	61
			Green 5.8	83	12	95
			WebbIE 3.14	112	13	125
			Enigma	33	17	50
		WININET.dll	IE Tab 2	151	4	155
			Lunascape 6	102	5	107
			Slim 5.01	148	3	151
			Green 5.8	138	6	144
			WebbIE 3.14	153	3	156
			Enigma	128	3	131
		MSHTML.dll	IE Tab 2	16	1	17
			Lunascape 6	16	1	17
			Slim 5.01	14	1	15
			Green 5.8	15	1	16
			WebbIE 3.14	16	1	17
			Enigma	16	1	17
Adobe Flash Player 11	open an ActionScript file	Flash11c.ocx	RealPlayer 14	40	8	48
			Winamp 5 Video	32	7	39
			IrfanView 4.27 plugin	21	3	24
			Microsoft PowerPoint 2010 (embedded SWF)	18	3	21
			Gom Player 2	11	2	13
			PotPlayer 1.5	24	6	30
	open a Flash Video file	Flash11c.ocx	RealPlayer 14	38	6	44
			Winamp 5 Video	44	6	50
			IrfanView 4.27 plugin	34	3	37
			Microsoft PowerPoint 2010 (embedded SWF)	30	7	37
			Gom Player 2	22	2	24
			PotPlayer 1.5	36	8	44
	open a Flash Audio file	Flash11c.ocx	RealPlayer 14	94	4	98
			Winamp 5	106	2	108
			IrfanView 4.27 plugin	26	1	27
			PotPlayer 1.5	32	1	33
QuickTime Player 7	open a QuickTime MOV file	QuickTime.qts	Internet Explorer 9	97	10	107
			Mozilla Firefox 7	98	8	106
			IrfanView 4.27 plugin	98	24	122
			RealPlayer 14	5	29	34
	open a QuickTime VR file	QuickTime.qts	Internet Explorer 9	1	18	19
			Mozilla Firefox 7	2	27	29
			IrfanView 4.27 plugin	505	0	505
			RealPlayer 14	2	40	42
Windows Live Mail 2011	render an HTML email	URLMON.dll	DreamMail 4	16	2	18
			IncrediMail	10	3	13
		WININET.dll	DreamMail 4	3	2	5
			IncrediMail	3	0	3
		MSHTML.dll	DreamMail 4	10	0	10
			IncrediMail	10	0	10
Microsoft WordPad	open an RTF file	msftedit.dll	Jarte	6	6	12

Table 5.1: Inconsistent policy configurations in test subjects w.r.t. given workloads.

Green Browser 5.8 [60], WebbIE 3.14 [156], and Enigma Browser [48]. The tested Trident-based browsers are in wide use. For example, IE Tab is among the most popular plugins for both Firefox and Chrome, and has millions of downloads and users [73, 74, 144], and Lunascape has more than 20 million downloads and millions of users.

We next describe the new security vulnerabilities we discovered. We have reported these problems to the affected software vendors, and Lunascape has already acknowledged our findings. Since we were able to manually trigger all these reported vulnerabilities, they constitute real, and some of which very serious, security concerns. We also provide further discussions in Section 5.4.5.

5.4.2 New Vulnerabilities Discovered

As discussed in Section 5.2, we can utilize our framework to detect the security vulnerability where the test subjects allow an insecure URL scheme that can be exploited by phishing attacks [79]. This section illustrates the effectiveness of our framework by detecting additional security vulnerabilities. Our high-level approach is as follows. We first capture inconsistent policy configurations from a given workload. Next we analyze them for detecting potential insecure component usage and then manually trigger them for validation. For evaluation, we consider the accesses to the URLs for Microsoft homepage and gmail account as workloads. Table 5.1 shows the inconsistent policy configurations our tool detected in the test subjects under the given workloads. Next we describe the security problems caused by them and our analysis approach.

Insecure Configuration of URL Security Zone

As a protection mechanism, IE categorizes URL namespaces into five types of URL security zones (*i.e.*, Local Intranet, Trusted Sites, Internet, Restricted Sites, and Local Machine). Each zone has a different trust level [2] to determine whether or not a URL action is allowed. For example, while Internet zone allows the execution of script code, Restricted Site zone does not.

This privilege-based protection mechanism can cause security vulnerabilities. Suppose that a web page on a particular zone accesses resources on less restrictive zones. In this case, when accessing the web page, privilege escalation happens, called *Zone Elevation*. This security vulnerability has been exploited by real-life attacks[1] based on Cross Zone Scripting [30]. To mitigate this issue,

[1] Examples include MS05-001, MS05-014, MS08-048, and CVE-2008-2281.

```
...
i1  push    2 // GET_FEATURE_FROM_PROCESS
i2  push    1 // FEATURE_ZONE_ELEVATION
i3  call    CoInternetIsFeatureEnabled
i4  cmp     eax, 1
i5  setnz   byte_6402C6C4
...
i6  cmp     byte_6402C6C4, 0
i7  jz      short loc_6397DAB1
...
```

(a) Zone elevation.

```
...
i1  mov  edi, offset g_FEATURE_ALLOW_LONG_INTERNATIONAL_FILENAMES
i2  push edi
i3  call CoInternetIsFeatureEnabledInternal
i4  neg  eax
i5  sbb  eax, eax
i6  inc  eax
i7  mov  _GlobalAllowLongIntlFileNames, eax
...
i8  cmp  _GlobalAllowLongIntlFileNames, 0
i9  jz   i10
...
```

(b) Long filename handling.

Figure 5.4: Policy configuration and evaluation.

IE blocks the Zone Elevation [81]. However, the test subjects do not block it and are vulnerable to these attacks. We next describe how to use our framework to detect these security vulnerabilities in the test subjects.

Under the gmail workload, `MSHTML.dll` stores the configuration of the security policy on Zone Elevation and checks it at runtime. Figure 5.4(a) illustrates its detailed code and operates as follows. First, `i1`–`i3` invoke an interface call `CoInternetIsFeatureEnabled` to `URLMON.dll`, which determines whether or not the current process enables a security policy `FEATURE_ZONE_ELEVATION` [82]. Then `i4`–`i5` initialize the memory buffer `byte_6402C6C4` based on the result of the function. The stored value is evaluated to check the configuration of the feature in `i6`–`i7`. In particular, if the feature is enabled, the conditional jump in `i7` falls through in the execution. Otherwise, it takes the jump.

For IE and the test subjects, we analyzed inconsistencies in `i7` to detect incorrect configurations. According to our analysis, only IE enables this feature, and the test subjects allow Zone Elevation. To validate our findings, we developed a trusted web site having an `<iframe>` tag to a local HTML

file. When accessing the site, we observed that the zone elevation is indeed only successful for the test subjects.

In order to launch the cross zone scripting attacks, it is necessary to run script code in a local HTML file. To block this malicious behavior, IE adopts a default protection mechanism, called Local Machine Zone Lockdown (LMZL) [80], which configures security policies on particular URL actions in a more restrictive manner. For example, IE disallows the execution of any script code in local HTML files by default. During our validation, we identified that the test subjects do not adopt LMZL, allowing the execution of local script code.

Thus, this insecure component usage can lead to serious security vulnerabilities. In particular, the disabled `FEATURE_ZONE_ELEVATION` and the missing LMZL make the test subjects vulnerable to cross zone scripting attacks.

Incorrect Handling of Long File Name

When IE 9 beta and the test subjects connect to `microsoft.com`, they configure and evaluate a security policy on `FEATURE_ALLOW_LONG_INTERNATIONAL_FILENAMES` at runtime. Figure 5.4(b) illustrates the relevant code of `WININET.dll` for this policy configuration and evaluation. In particular, `i1`–`i6` determine whether or not the feature is enabled by invoking `CoInternetIsFeatureEnabledInternal`. Later `i7` writes the result to the memory buffer `_GlobalAllowIntlFileNames` in the global data region of `WININET.dll`. For evaluation, `i8` reads the configuration data from the `_GlobalAllowIntlFileNames`, and `i9` evaluates the configuration by comparing its value of the data with zero. According to our analysis, the test subjects take the branch at `i9`, but IE 9 beta falls through. This shows that only IE 9 beta enables the feature.

This feature is related to the maximum path length limitation [115]. In particular, for a given file, its fullpath length cannot be longer than 256. Suppose that IE downloads and opens a file whose name having non-ASCII characters. In this case, the previous IE releases store the file to the temporary folder by encoding its name based on UTF-8 [150] and opens it based on the encoded fullpath. However, the length of its fullpath is often longer than the given limit. For example, when a `.xlsx` file whose name is composed of 17 ASCII and 12 Korean characters is downloaded, the length of its encoded fullpath is larger than 256. In this case, Microsoft Excel 2010 cannot open the downloaded file [165]. To mitigate this issue, Microsoft released a hotfix KB982381, and recent

IE releases changed the encoding scheme for international file names. We confirmed the detected inconsistent configuration because the test subjects cannot download and open the `.xlsx` file. Thus, incorrect policy configurations may also cause compatibility issues.

5.4.3 Root-Cause Analysis of Newly Discovered Vulnerabilities

As we discussed in Section 5.4.2, the insecure configuration of the URL zones leads to security vulnerabilities. To mitigate this issue, it is necessary to configure and evaluate the URL action policies in a secure manner. However, we observe that third-party developers often insecurely reuse the IE browser components without considering this issue, which makes the test subjects disable these protection mechanisms of IE. For example, although IE supports XSS and Phishing filters [71, 136] by default, the test subjects neither configure relevant security policies nor block the malicious behavior. To address this problem, it is necessary to understand the root cause of this insecurity. In this section, we present how to use of our framework to analyze insecure URL action policies.

Evaluation of URL Action Policies

According to MSDN, the evaluation of URL action policies is performed by certain interface calls exported by `URLMON.dll`. For example, `ProcessUrlAction` [76] determines whether or not a specified action for a particular URL is allowed. Based on this information, we reverse engineered such interface calls to analyze the detailed process of evaluating URL action policies.

Figure 5.5 shows the high-level overview of this evaluation. In particular, `URLMON.dll` maintains URL action policies as a list of memory buffers in its global data region. Each buffer contains information on a URL zone, a URL action (*e.g.*, downloading signed ActiveX), and its policy (*e.g.*, allow). The evaluation of the URL action policies is performed by an internal function, which is invoked by several interface calls at runtime. In particular, the function takes three parameters (*i.e.*, a URL zone, a URL action, and a policy to check) and iterates through the memory buffers to locate the configuration whose data are matched with the parameters. If such a memory buffer is found, the function returns true, showing that the specified URL action policies are matched with the current configuration setting. To extract the URL action polices checked by the reference and the test subjects at runtime, we can use our framework to infer the detailed information on policy evaluation.

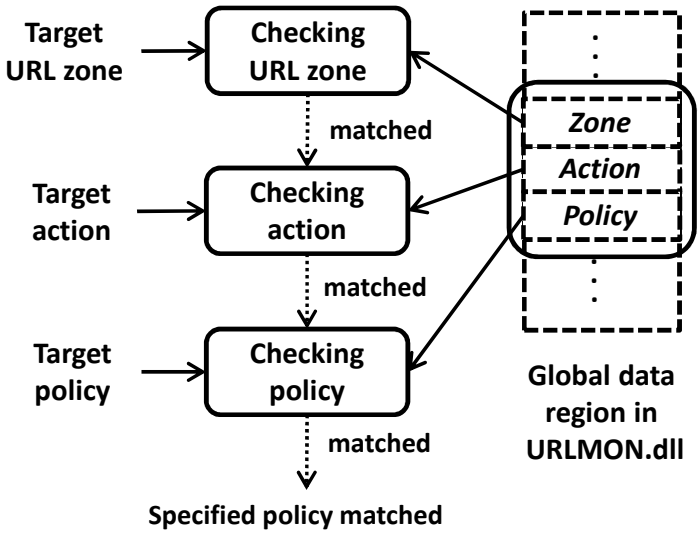


Figure 5.5: Evaluation of URL action policies.

This is because 1) the configuration data is stored in the global memory space, and 2) matching the parameters with the configuration data is performed by conditional jumps affected by the data read.

Based on the above observations, we analyze the runtime traces obtained in the second phase of Figure 5.1 to extract the evaluations performed by the reference and the test subjects. To this end, we first locate the matched conditional jumps on the target policy. Then we traverse the traces backward to find the consecutive conditional jumps that check the target action and URL zone. According to Figure 5.5, the data reads for evaluating the detected conditional jumps correspond to the target URL zone, action, and its policy. Using this approach, we can extract the evaluations of URL action policies performed by the reference and the test subjects. In the following sections, we discuss and analyze the security vulnerabilities caused by insecure configurations of URL action policies.

Disabled XSS and Phishing Filters

Recent IE releases have supported the XSS filter [71] and the Phishing (or SmartScreen) filter [136] by default. These mechanisms can effectively protect users from unknown XSS and phishing attacks. We confirmed that the test subjects do not enable these protection mechanisms even though they use the same browser components. To analyze these security vulnerabilities, we access malicious web sites that trigger these filters (*i.e.*, our workloads). Then we apply our framework to

URL action	XSS filter (Internet)		Phishing filter (Internet)		Script execution (Local machine)	
	Reference	Test subjects	Reference	Test subjects	Reference	Test subjects
URLACTION_DOWNLOAD_SIGNED_ACTIVEX	✓	✗	✓	✗	✗	✗
URLACTION_DOWNLOAD_UNSIGNED_ACTIVEX	✓	✗	✓	✗	✗	✗
URLACTION_ACTIVEX_OVERRIDE_OBJECT_SAFETY	✓	✓	✓	✗	✗	✗
URLACTION_SCRIPT_RUN	✓	✓	✓	✓	✓	✗
URLACTION_SCRIPT_XSSFILTER	✓	✗	✗	✗	✗	✗
URLACTION_HTML_INCLUDE_FILE_PATH	✓	✗	✗	✗	✗	✗
URLACTION_SHELL_VERB	✓	✗	✓	✗	✗	✗
URLACTION_SHELL_EXECUTE_HIGHRISK	✓	✗	✓	✗	✗	✗
URLACTION_COOKIES_ENABLED	✓	✓	✗	✗	✗	✗
URLACTION_BEHAVIOR_RUN	✓	✓	✓	✗	✗	✗
URLACTION_FEATURE_MIME_SNIFFING	✓	✗	✓	✗	✗	✗
URLACTION_FEATURE_DATA_BINDING	✓	✓	✓	✓	✓	✗
URLACTION_ALLOW_APEVALUATION	✗	✗	✓	✗	✓	✗
URLACTION_INPRIVATE_BLOCKING	✓	✓	✓	✓	✓	✗
URLACTION_ALLOW_STRUCTURED_STORAGE_SNIFFING	✗	✗	✓	✓	✗	✗

Table 5.2: Evaluated URL action policies for XSS and Phishing filters / local script execution.

capture the runtime traces of the reference and the test subjects running under the given set of workloads. We next extract the evaluations of these URL action policies using our approach discussed in Section 5.4.3. Table 5.2 shows the evaluated URL action policies for the Internet zone under our workloads, where ✓ represents the case that the corresponding policy has been evaluated, and ✗ represents the case that the corresponding policy has not been evaluated. It is interesting that the URL action policies relevant to XSS and Phishing filters are evaluated only when these filters are enabled. This information helps us pinpoint the code relevant to the evaluations of the policies.

In the case of XSS filter, `MSHTML.dll` calls an internal function `IsXssFilterEnabled`, which invokes an external function `ProcessUrlAction` exported by `URLMON.dll`, to check whether or not the XSS filter is enabled. Considering the Phishing filter, `MSHTML.dll` calls an internal function `CMarkup::ProcessUrlAction2` to invoke `ProcessUrlActionEx2` to `URLMON.dll`, which checks whether or not the Phishing filter is enabled. The information on the caller-callee relationship can be a starting point for analyzing software behavior relevant to these URL actions. Note that the top of the stack at the interface entries contains the address to be returned after invoking the interface call.

Disabled Local Machine Zone Lockdown

As we have discussed in Section 5.4.2, the test subjects allow local script execution, making them vulnerable to cross zone scripting attacks. To analyze this security vulnerability, we run local script code using an external script file as workload and repeat the analysis steps in Section 5.4.3.

Workload	Target component	IP analysis (s)	Runtime trace extraction (s)							Offline phase (s)
			IE 9	IE Tab 2	Lunascape 6	Slim 5.0.1	Green 5.8	WebbIE 3.14	Enigma	
microsoft.com	URLMON.dll	73.8	169.7	264.6	792.0	865.4	354.7	444.8	288.5	152.2
	WININET.dll	108.2	265.6	483.9	290.1	165.2	247.0	300.2	267.2	237.7
gmail account	URLMON.dll	201.1	249.0	197.7	221.4	159.9	258.4	238.8	188.8	518.5
	WININET.dll	270.5	583.1	348.0	327.0	315.8	282.8	384.0	349.2	536.5
	MSHTML.dll	1,852.2	1,902.2	1,909.2	1,774.7	1,698.6	1,462.1	1,774.2	1,436.0	59.6
XSS filter	URLMON.dll	82.3	79.8	111.6	92.5	142.9	78.4	74.7	117.2	106.0
Phishing filter	URLMON.dll	76.5	104.6	179.6	89.3	75.8	71.4	71.3	79.8	98.6
Local script run	URLMON.dll	190.9	105.5	67.1	86.9	83.7	67.7	79.6	66.5	89.3

Table 5.3: Execution time for each analysis phase.

Table 5.2 shows the evaluated URL action policies for Local Machine during the workload. According to our result, all test subjects execute the local script code without evaluating any security policy on its action. However, IE evaluates the security policies on the potentially malicious behavior and blocks it. For example, IE blocks the execution of local script code, protecting IE from the cross zone scripting attacks. Similar to the Phishing filter case, the function `CMarkup::ProcessUrlAction2` of `MSHTML.dll` invokes `ProcessUrlActionEx2` to evaluate the security policy on `URLACTION_SCRIPT_RUN` at runtime.

5.4.4 Performance

Table 5.3 shows the execution time for each phase of our analysis discussed in Section 5.4.2. All experiments were done on a Core2 Duo 2.40GHz processor with 4GB RAM. The results show that our framework can easily scale to the analysis of real-world applications. For example, it can detect, in about two hours, inconsistent policy configurations of the three components by the six real-world IE-based browsers accessing a complex web sites such as `microsoft.com`.

Detecting insecure component usage from the gmail workload is relatively more time consuming. In particular, the analysis of `MSHTML.dll` took about four hours. The main reasons are as follows. First, the user login is necessary to access the gmail account, and `MSHTML.dll` is heavily used for this [78]. Second, when an instruction is executed, the second phase in our framework checks whether or not the instruction is to be instrumented. Because `MSHTML.dll` is a large file whose size is about 12MB, a large number of checking is necessary, even though the number of instrumented instructions is relatively small. Despite the additional performance overhead, the analysis time for each browser is reasonable. For example, the analysis of `MSHTML.dll` used by IE Tab 2 took about

thirty minutes.

It is interesting to note that the offline analysis phase for `MSHTML.dll` is relatively fast. This is because the size of the traces extracted is much smaller. For example, while the analysis on `URLMON.dll` generates runtime traces of 335MB during the gmail workload, the analysis on `MSHTML.dll` only generates runtime traces of 7.53 MB. The main reason is that the accesses to the global data region of `MSHTML.dll` is rare at runtime. For the gmail workload performed by IE Tab 2, `URLMON.dll` and `MSHTML.dll` access their global data regions 800,120 times and 1,585 times, respectively.

5.4.5 Further Discussions and Analysis

We now further discuss two natural questions: 1) Are the other detected inconsistent policy configurations security relevant? and 2) How about Gecko- and WebKit-based Browsers?

Analyzing the Other Detected Inconsistencies. Our framework provides useful information on inconsistent configurations and evaluations of security polices. As we have discussed, such information is effective at detecting and analyzing insecure component usage. However, the detected inconsistent executions may not all be security relevant. For example, `WININET.dll` maintains a configuration of Autodial [8] in its global data region, and only IE enables it. Also, while IE initializes User-Agent String [147] whose configuration is stored in `URLMON.dll`, the test subjects do not. Although these configurations are inconsistent, they may not introduce security issues for the test subjects. However, such information can still be valuable for improving the functionalities of the client software using the components.

To determine the importance of the detected inconsistencies, domain knowledge of the test subjects is typically needed. Sometimes we are not able to determine whether or not the detected inconsistencies may cause security vulnerabilities. For example, an internal function of `URLMON.dll` takes the data stored in the global data region as a parameter, and a conditional jump is affected by the return value of the function. We observed that IE falls through and the test subjects take the branch at the conditional jump. Although this inconsistency may lead to a security problem, it is difficult for us to analyze it as we do not know the precise semantics of this function.

Also, the inconsistent policy configurations detected from the non-IE components (such as Adobe Flash Player and QuickTime Player) can also lead to security vulnerabilities. However,

because we lack domain-expert knowledge on these components, we have focused our analysis on the IE components. In particular, the source code and the detailed documentation for the non-IE components are not publicly available. On the other hand, our results have clearly demonstrated that inconsistent policy configuration is a general concern that affects many applications.

Analyzing Gecko- and WebKit-based Browsers. Besides Trident-based browsers, Gecko- and WebKit-based browsers, such as Firefox and Google Chrome, are also widely used. This section analyzes and discusses their usage of the Gecko and WebKit engines.

Gecko [52] is an open source browser engine developed by Mozilla. It is used as the HTML rendering engine by many Mozilla and third-party browsers; notable ones include Firefox, SeaMonkey, and Lunascape [53]. Perhaps less known is that Gecko-based browsers do not reuse the same *binary* Gecko components. Instead, Gecko's source code is modified and reused. First, developers often adapt the same source code of the engine for use in their own applications. For example, both Firefox 5.0 and SeaMonkey 2.2 are based on Gecko 5.0, but their versions of `xul.dll` are modified and different. Second, developers may choose to use different versions of the engine. For example, Firefox 5.0 and Lunascape 6.5, the latest releases of the browsers as of July 2011, use Gecko 5.0 and 1.9.2, respectively [53]. Different versions of the same component can be significantly different. The size of `xul.dll` for example has increased from 90KB to more than 13MB.

For evaluation, we chose two commonly used, security relevant Gecko components, `nss3.dll` and `ssl3.dll`, and analyzed their usage by Firefox 5.0 (reference) and Lunascape 6.5 (test subject). We used the same workloads as shown in Table 5.1. Our evaluation results indicate that the two components are used consistently by Firefox and Lunascape for the given workloads.

WebKit [142] is another open source browser engine that has been used by many browsers. Although it is well-known that both Google Chrome and Safari *use* WebKit, how it is used is quite different. Chromium is an open source version of Google Chrome, and developers have adapted the WebKit code for their own use [68]. In addition, the rendering code is not an independent component, but part of `chrome.dll`, the large main executable of Google Chrome. In particular, `chrome.dll` contains the code for web browsing and is over 25MB. In contrast, Safari uses WebKit for rendering (*i.e.*, `WebKit.dll`) and dynamically uses the component.

5.5 Related Work

This section surveys closely related work, which we divide into four categories: bug detection via inconsistent software behavior detection, detection of component insecurity, framework for secure component usage, and detection of violated browser policies.

This section surveys closely related work, which we divide into four categories: bug detection via inconsistent software behavior detection, detection of component insecurity, frameworks for secure component usage, and detection of violated browser policies.

Bug Detection via Inconsistent Software Behavior Detection. Brumley *et al.* [25] present a bug detection technique to discover deviations among different implementations of the same protocol specification. The technique is related to ours because it is also based on the general differential analysis concept. It analyzes different software and detects inconsistent behavior that is supposed to be consistent. However, the technique has a different goal from ours. Their goal is to detect inconsistent implementations of the same protocol, while ours is to detect inconsistent policy evaluations that may lead to insecure component reuse.

Detection of Component Insecurity. The detection of insecure software components has been actively studied. Neuhaus *et al.* [118] propose a technique that performs statistical analysis on vulnerability history; the function calls and the imports of each vulnerable component are utilized to characterize the corresponding vulnerabilities. Bandhakavi *et al.* [15] present a static analysis to detect information flow vulnerabilities in Firefox extensions. Dhawan *et al.* [41] dynamically track the execution of JavaScript extensions in Firefox to detect information flow violations. Guha *et al.* [63] statically check security of the browser extensions by using software verification techniques. In comparison, while these techniques detect the insecurity of target components, we focus on detecting *insecure usage* of the components. In particular, we model insecure usage of a component as inconsistent evaluations of security policies maintained by the component. With this model, we are able to, for example, detect and analyze browser components insecurely used by IE-based browsers (see Section 5.4).

Framework for Secure Component Usage. Because malicious or vulnerable components can introduce security problems to software, extensive research has been conducted on protecting software

against them. For example, secure browsers [62, 139, 153] apply sandboxing techniques to protect them from crashed plugins. Grier *et al.* [61] present security policies to use browser plugins in a secure manner. Barth *et al.* [18] propose a technique to mitigate the damage caused by the exploitation of vulnerable extensions by designing least privilege, privilege separation, and strong isolation. Kirda *et al.* [88] detect malicious browser components by monitoring spyware-like behavior. These techniques aim at detecting and protecting against insecure execution of target components, while our purpose is to detect insecure usage of components.

Detection of Violated Browser Policies. A browser's security policy serves as a key part for safe web browsing. Thus, modern browsers support a number of policies to improve their security [24]. Based on this insight, many researchers [17, 19–21, 83] have focused on detecting violations of browser security policies. Although our framework also detects the violation of browser security policies (Section 5.4), its goal is different from that of these previous techniques. We aim at detecting security policies incorrectly configured by insecure browser component usage. In contrast, the aforementioned techniques detect subversions of the enforced security policies.

5.6 Conclusion and Future Work

We have presented an effective framework to detect and analyze insecure component usage. Our key idea is to detect inconsistent security policy configurations. Suppose that both a reference and a test subject use a component that maintains the configuration of a security policy. If they use the component in ways that make the policy inconsistently evaluated, the test subject can be vulnerable to attacks intended to be blocked by the policy. We model component usage relevant to the policy as memory access patterns and the conditional jumps affected by them. Based on this model, we have presented a program analysis technique to locate inconsistent policy configurations at runtime. Our evaluation results show that our technique is effective at detecting and analyzing insecure component usage. In particular, it detected inconsistent policy configurations of real-world applications and discovered several new security vulnerabilities of IE-based browsers. We have also shown that our framework can be used effectively to conduct detailed analysis of security vulnerabilities related to insecure component usage.

For future work, we would like to analyze insecure usage of other widely-used components.

Our current implementation focuses on analyzing the global data region to detect component usage relevant to security policies. We plan to expand the work's scope by handling other types of non-thread specific memory regions (*e.g.*, the heap) to detect and analyze general inconsistent component usage.

Chapter 6

Conclusion

This dissertation has formulated insecure component integration and empirically evaluated its effect on software security. In particular, we have examined *unsafe component loading* and *insecure component usage*, and have developed effective techniques to detect and analyze them. This chapter summarizes the dissertation and discusses future research directions.

6.1 Summary

Unsafe Component Loading and Its Detection (Section 2.1 and Chapters 3–4). Dynamic loading of software components (*e.g.*, libraries or module) is a widely used mechanism for improved productivity and reliability. Correct component resolution is critical for reliable and secure software execution. However, programming mistakes may lead to unintended or even malicious components to be resolved and loaded. In particular, dynamic loading can be hijacked by placing an arbitrary file with the specified name in a directory searched before resolving the target component.

We have presented the first automated technique to detect unsafe component loadings. We cast our technique as two phases: 1) apply dynamic binary instrumentation to collect runtime information on component loading (*online phase*); and 2) analyze the collected information to detect vulnerable component loadings (*offline phase*). For evaluation, we implemented our technique to detect unsafe component loadings in popular software on Microsoft Windows and Linux. Our evaluation results show that unsafe component loading is prevalent in software on both OS platforms, and it is more severe on Microsoft Windows. In particular, our tool detected more than 4,000 unsafe

component loadings in our evaluation, and some lead to remote code execution on Microsoft Windows. The issues uncovered in our work and two related efforts had also been the topic of extensive media coverage in August 2010.

Although our dynamic detection is simple and effective in detecting unsafe component loadings, it suffers from limited code coverage and may miss important vulnerabilities. Thus, it is desirable to develop effective techniques to detect *all possible* unsafe component loadings. To this end, we present the first static binary analysis aiming at detecting all possible loading-related errors. The key challenge is how to scalably and precisely compute what components may be loaded at relevant program locations. Our main insight is that this information is often determined locally from the component loading call sites. This motivates us to design a demand-driven analysis, working backward starting from the relevant call sites. In particular, for a given call site c, we first compute its *context-sensitive executable slices*, one for each execution context. Then we emulate the slices to obtain the set of components possibly loaded at c. This novel combination of slicing and emulation achieves good scalability and precision by avoiding expensive symbolic analysis. We implemented our technique and evaluated its effectiveness against the existing dynamic technique on nine popular Windows applications. Results show that our tool has better coverage and is precise—it is able to detect many more unsafe loadings. It is also scalable and finishes analyzing all nine applications within minutes.

Insecure Component Usage and Its Detection (Section 2.2 and Chapter 5). The lack of expert knowledge can make developers utilize a component in an insecure manner. For example, we noticed that common IE-based browsers, such as IE Tab, disable important security features that IE enables by default, even though they all use the same browser components. This insecure usage renders these IE-based browsers vulnerable to the attacks blocked by IE. To our knowledge, this important security aspect of component reuse has largely been unexplored.

We have presented the first practical framework for *detecting and analyzing vulnerabilities of insecure component usage*. Its goal is to enforce and support secure component reuse. Our core approach is as follows. Suppose that component C maintains a security policy configuration to block certain malicious behavior. If two clients of component C, say a reference and a test subject, handle the malicious behavior inconsistently, the test subject uses C insecurely. In particular, we

model component usage related to a policy based on (1) accesses to the configuration state inside the component and (2) the conditional jumps affected by the data read from the state. We utilize this model to detect inconsistent policy evaluations, which can lead to insecure component usage. We have implemented our technique for Windows applications and used it to detect and analyze insecure usage of popular software components. Our evaluation results show that our framework is scalable and effective at detecting and analyzing insecure component usages. In particular, we detected several serious, previously unknown vulnerabilities in Internet Explorer 9 and helped perform detailed analysis of insecure component usage.

6.2 Future Work

For future work, it would be interesting to consider the following two research directions: (1) *investigating other types of insecure component integration*, and (2) *exploring security issues in source code reuse*.

Other Types of Insecure Component Integration. We have focused on examining *unsafe component loading* and *insecure component usage*. Although it is evident that they can lead to security vulnerabilities, there may exist other issues of insecure component integration. Thus, it would be interesting to investigate security vulnerabilities caused by other types of insecure component integration and evaluate their prevalence and severity in real-life software.

Security Issues in Source Code Reuse. In Chapter 5, we have shown how to detect insecure reuse of *binary* components. Although software generally reuses desired components based on their binary releases, source code adaptation and reuse are also common. Specifically, developers may reuse or customize the source code of a component to better fit their needs. For example, the source code of Gecko and the WebKit components have been reused for rendering HTML documents in a number of popular software projects. Although source code reuse is very common, its security impact has not been much explored. It would be interesting to systematically explore this direction, similar to what we have done in this dissertation for binary components.

Bibliography

[1] About the security content of Safari 3.1.2 for Windows. http://support.apple.com/kb/HT2092.

[2] About URL Security Zones. http://msdn.microsoft.com/en-us/library/ms537183(v=vs.85).aspx.

[3] About Windows Resource Protection. http://msdn.microsoft.com/en-us/library/aa382503(VS.85).aspx.

[4] About Windows Resource Protection. http://www.dependencywalker.com/help/html/dependency_types.htm.

[5] Ákos Kiss, Judit Jász, Gábor Lehotai, and Tibor Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings of the 3rd IEEE International Workshop on Source Code Analysis and Manipulation*, 2003.

[6] An update on the DLL-preloading remote attack vector. http://blogs.technet.com/b/srd/archive/2010/08/31/an-update-on-the-dll-preloading-remote-attack-vector.aspx.

[7] Shay Artzi, Adam Kiezun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.

[8] Autodial. http://technet.microsoft.com/en-us/library/cc781180(W.10).aspx.

[9] Algirdas Avizienis. Toward systematic design of fault-tolerant systems. *Computer*, 30:51–58, April 1997.

[10] AxFuzz. `http://sourceforge.net/projects/axfuzz/`.

[11] AxMan. `http://www.metasploit.com/users/hdm/tools/axman/`.

[12] Michael Bailey, Jon Oberheide, Jon Andersen, Z. Morley Mao, Farnam Jahanian, and Jose Nazario. Automated Classification and Analysis of Internet Malware. In *Proceedings of the 10th Recent Advances in Intrusion Detection*, 2007.

[13] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 13th International Conference on Compiler Construction*, 2004.

[14] Gogul Balakrishnan and Thomas Reps. Analyzing stripped device-driver executables. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[15] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. Vex: vetting browser extensions for security vulnerabilities. In *Proceedings of the 19th USENIX Security Symposium*, 2010.

[16] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36:81–94, November 1993.

[17] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, 2009.

[18] Adam Barth, Adrienne Porter Felt, Prateek Saxena, and Aaron Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the 17th Network and Distributed System Security Symposium*, 2009.

[19] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, 2008.

[20] Adam Barth, Collin Jackson, and John C. Mitchell. Securing frame communication in browsers. *Commun. ACM*, 52, June 2009.

[21] Adam Barth, Joel Weinberger, and Dawn Song. Cross-origin javascript capability leaks: detection, exploitation, and defense. In *Proceedings of the 18th conference on USENIX security symposium*, 2009.

[22] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the 16th Symposium on Network and Distributed System Security*, 2009.

[23] David Binkley. Precise executable interprocedural slices. *ACM Lett. Program. Lang. Syst.*, 2(1-4):31–45, 1993.

[24] Browser Security Handbook. `http://code.google.com/p/browsersec/wiki/Main`.

[25] David Brumley, Juan Caballero, Zhenkai Liang, Newso James, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[26] David Brumley, Dawn Xiaodong Song, Tzi-cker Chiueh, Rob Johnson, and Huijia Lin. RICH: Automatically protecting against integer-based vulnerabilities. In *In Proceedings of the 14th Network and Distributed System Security Symposium*, 2007.

[27] Cristian Cadar, Daniel Dunbar, and Dawson 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*, OSDI, 2008.

[28] Cristian Cadar, Daniel Dunbar, and Dawson 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*, 2008.

[29] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[30] CAPEC-104: Cross Zone Scripting. `http://capec.mitre.org/data/definitions/104.html`.

[31] Suresh Chari, Shai Halevi, and Wietse Venema. Where do you want to go today? escalating privileges by pathname manipulation. In *In Proceedings of the 17th Network and Distributed System Security Symposium*, 2010.

[32] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, 2005.

[33] Mihai Christodorescu, Somesh Jha, and Christopher Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007.

[34] Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 26th IEEE Symposium on Security and Privacy*, 2005.

[35] Cristina Cifuentes and Antoine Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings of the 23rd International Conference on Software Maintenance*, 1997.

[36] COMBust. `http://www.blackhat.com/presentations/bh-usa-03/bh-us-03-bretmounet-combust.zip`.

[37] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying dormant functionality in malware programs. In *Proceedings of the 31st IEEE Symposium on Security and Privacy*, 2010.

[38] COMRaider. `http://labs.idefense.com/software/fuzzing.php#more_comraider`.

[39] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684 –702, sept.-oct. 2009.

[40] Premkumar T. Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, 2000.

[41] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in javascript-based browser extensions. *Computer Security Applications Conference, Annual*, 2009.

[42] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering*, 2006.

[43] dlopen man page. `http://linux.die.net/man/3/dlopen`.

[44] Dranzer. `http://www.cert.org/vuls/discovery/dranzer.html`.

[45] Dynamic-Link Library Search Order. `http://msdn.microsoft.com/en-us/library/ms682586(VS.85).aspx`.

[46] Dynamic-Link Library Security. `http://msdn.microsoft.com/en-us/library/ff919712(VS.85).aspx`.

[47] Dynamic Loading. `http://en.wikipedia.org/wiki/Dynamic_loading`.

[48] Enigma Browser. `http://www.suttondesigns.com/`.

[49] Executable and Linkable Format (ELF). `http://www.skyfree.org/linux/references/ELF_Format.pdf`.

[50] Exploiting DLL Hijacking Flaws. `http://blog.metasploit.com/2010/08/exploiting-dll-hijacking-flaws.html`.

[51] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. Syst.*, 9(3):319–349, 1987.

[52] Gecko. `https://developer.mozilla.org/en/Gecko`.

[53] Gecko (Layout Engine). http://en.wikipedia.org/wiki/Gecko_(layout_engine).

[54] GNU C Library. http://www.gnu.org/software/libc.

[55] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[56] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *Proceedings of the 15th Network and Distributed System Security Symposium*, 2008.

[57] Patrice Godefroid, Michael Y. Levin, and David A Molnar. Automated whitebox fuzz testing. In *In Proceedings of the 15th Network and Distributed System Security Symposium*, 2008.

[58] Karen Mercedes Goertzel, Theodore Winograd, and Booz Allen Hamilton. Safety and security considerations for component-based engineering of software-intensive systems. Technical report, https://buildsecurityin.us-cert.gov/swa/downloads/NAVSEA-Composition-DRAFT-061110.pdf, 2010.

[59] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the 6th USENIX Security Symposium*, 1996.

[60] Green Browser 5.8. http://greenbrowser.en.softonic.com/.

[61] Chris Grier, Samuel T. King, and Dan S. Wallach. How I learned to stop worrying and love plugins. In *Web 2.0 Security and Privacy*, 2009.

[62] Chris Grier, Shuo Tang, and Samuel T. King. Secure Web browsing with the OP Web browser.

[63] Arjun Guha, Matthew Fredrikson, Benjamin Livshits, and Nikhil Swamy. Verified security for browser extensions. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[64] Hacking Toolkit Publishes DLL Hijacking Exploit. http://www.computerworld.com/s/article/9181513/Hacking_toolkit_publishes_DLL_hijacking_exploit.

[65] Thorsten Holz, Felix Freiling, and Carsten Willems. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2), 2007.

[66] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Prog. Lang. Syst.*, 12(1):26–60, 1990.

[67] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the 3rd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1995.

[68] How Chromium Displays Web Pages. http://www.chromium.org/developers/design-documents/displaying-a-web-page-in-chrome.

[69] IDA Pro Disassembler. http://www.hex-rays.com/idapro/.

[70] IDAPython. http://code.google.com/p/idapython/.

[71] IE Cross-site Scripting Filter. http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/cross-site-scripting-filter.

[72] IE Tab 2. https://addons.mozilla.org/en-US/firefox/addon/ie-tab/.

[73] IE Tab for Chrome. http://www.chromeextensions.org/utilities/ie-tab/.

[74] IE Tab for Firefox. https://addons.mozilla.org/en-US/firefox/addon/ie-tab/.

[75] IE's unsafe DLL loading. http://www.milw0rm.com/exploits/2929.

[76] IInternetSecurityManager::ProcessUrlAction Method. http://msdn.microsoft.com/en-us/library/ms537136(v=vs.85).aspx.

[77] Insecure Library Loading Could Allow Remote Code Execution. http://www.microsoft.com/technet/security/advisory/2269637.mspx.

[78] Internet Explorer Architecture. http://msdn.microsoft.com/en-us/library/aa741312(28v=vs.85)29.aspx.

[79] Internet Explorer does not support user names and passwords in Web site addresses (HTTP or HTTPS URLs) . http://support.microsoft.com/kb/834489.

[80] Internet Explorer Local Machine Zone Lockdown. http://technet.microsoft.com/en-us/library/cc782928(WS.10).aspx.

[81] Internet Explorer Zone Elevation Blocks. http://technet.microsoft.com/en-us/library/cc757200(WS.10).aspx.

[82] Introduction to Feature Controls. http://msdn.microsoft.com/en-us/library/ms537184(v=vs.85).aspx.

[83] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from dns rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[84] Killbit. http://support.microsoft.com/kb/240797.

[85] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[86] Samuel T. King and Peter M. Chen. Backtracking Intrusions. In *Proceedings of the 19th Symposium on Operating Systems Principles*, 2003.

[87] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[88] Engin Kirda, Christopher Kruegel, Greg Banks, Giovanni Vigna, and Richard A. Kemmerer. Behavior-based spyware detection. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[89] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and Xiaofeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of the 18th conference on Usenix Security Symposium*, 2009.

[90] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[91] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Proceedings of the 8th Symposium on Recent Advances in Intrusion Detection*, 2005.

[92] Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.

[93] Christopher Kruegel, William Robertson, and Giovanni Vigna. Static analysis of executables to detect malicious patterns. In *Proceedings of the 20th Annual Computer Security Applications Conference*, 2004.

[94] Taeho Kwon and Zhendong Su. Automatic detection of unsafe component loadings. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, 2010.

[95] David Larochelle and David Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[96] ld-linux man page. `http://linux.die.net/man/8/ld-linux`.

[97] Tony Lee and Jigar J. Mody. Behavioral Classification. In *Proceedings of the 15th EICAR Annual Conference*, 2006.

[98] libdvdcss. `http://en.wikipedia.org/wiki/Libdvdcss`.

[99] LoadLibraryEx Function. `http://msdn.microsoft.com/en-us/library/ms684179(v=vs.85).aspx`.

[100] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[101] Lunascape 6. `http://www.lunascape.tv/`.

[102] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[103] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[104] William M. Mckeeman. Differential testing for software. *Digital Technical Journal*, 10(1):100107, 1998.

[105] Microsoft Cooking Up Baker's Dozen of Fixes for Patch Tuesday. `http://www.esecurityplanet.com/patches/article.php/3902856/Microsoft-Cooking-Up-Bakers-Dozen-of-Fixes-for-Patch-Tuesday.htm`.

[106] Microsoft Portable Executable and Common Object File Format Specification. `http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx`.

[107] Microsoft releases tool to block DLL load hijacking attacks. `http://www.computerworld.com/s/article/print/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks`.

[108] Microsoft releases tool to block DLL load hijacking attacks. `http://www.computerworld.com/s/article/9181518/Microsoft_releases_tool_to_block_DLL_load_hijacking_attacks`.

[109] Microsoft Security Bulletin MS09-014. `http://www.microsoft.com/technet/security/Bulletin/MS09-014.mspx`.

[110] Microsoft Security Bulletin MS09-015. `http://www.microsoft.com/technet/security/Bulletin/MS09-015.mspx`.

[111] Microsoft Was Warned of DLL Vulnerability a Year Ago. `http://www.esecurityplanet.com/features/article.php/3900186/Microsoft-Was-Warned-of-DLL-Vulnerability-a-Y ear-Ago.htm`.

[112] David Molnar, Xue Cong Li, and David A. Wagner. Dynamic test generation to find integer bugs in x86 binary Linux programs. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[113] Alexander Moshchuk, Tanya Bragin, Steven D. Gribble, and Henry M. Levy. A crawler-based study of spyware on the web. In *Proceedings of the 13th Annual Network and Distributed Systems Security Symposium*, 2006.

[114] MS09-014: Addressing the Safari Carpet Bomb vulnerability. `http://blogs.technet.com/srd/archive/2009/04/14/ms09-014-addressing-the-safari-carpet-bomb-vuln erability.aspx`.

[115] Naming Files, Paths, and Namespaces. `http://msdn.microsoft.com/en-us/library/aa365247%28v=vs.85%29.aspx`.

[116] NetworkX. `http://networkx.lanl.gov/`.

[117] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[118] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[119] Alessandro Orso, Saurabh Sinha, and Mary Jean Harrold. Incremental slicing based on data-dependence types. In *Proceedings of the 27th IEEE International Conference on Software Maintenance*, 2001.

[120] pefile. `http://code.google.com/p/pefile/`.

[121] PyEmu. `http://code.google.com/p/pyemu/`.

[122] Thomas Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the 5th International Conference on Compiler Construction*, 1994.

[123] Thomas Reps and Gogul Balakrishnan. Improved memory-access analysis for x86 executables. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*, 2008.

[124] Thomas Reps, Susan Horwitz, Mooly Sagiv, and Genevieve Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, 1994.

[125] Researcher told Microsoft of Windows apps zero-day bugs 6 months ago. `http://www.computerworld.com/s/article/print/9181358/Researcher_told_Microsoft_of_Windows_apps_zero_day_bugs_6_months_ago`.

[126] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dussel, and Pavel Laskov. Learning and Classification of Malware Behavior. In *Proceedings of the 5th Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[127] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *In Proceedings of the 11th Network and Distributed System Security Symposium*, 2004.

[128] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the 18th International Symposium on Software Testing and Analysis*, 2009.

[129] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the 8th International Symposium on Software Testing and Analysis*, 2009.

[130] Edward J. Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proceedings of the 31th IEEE Symposium on Security and Privacy*, 2010.

[131] Koushik Sen, Darko Marinov, and Gul Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2005.

[132] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[133] Side-by-side Assemblies. `http://msdn.microsoft.com/en-us/library/aa376307(VS.85).aspx`.

[134] Saurabh Sinha, Mary Jean Harrold, and Gregg Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, 1999.

[135] Slim Browser 5.01. `http://www.slimbrowser.net/en/`.

[136] SmartScreen Filter. `http://windows.microsoft.com/en-US/internet-explorer/products/ie-9/features/smartscreen-filter`.

[137] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. BitBlaze: A new approach to computer security via binary analysis. In *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[138] Source lines of code. `http://en.wikipedia.org/wiki/Source_lines_of_code`.

[139] Shuo Tang, Haohui Mai, and Samuel T. King. Trust and protection in the illinois browser operating system. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, 2010.

[140] The End of DLL Hell. `http://msdn.microsoft.com/en-us/library/ms811694.aspx`.

[141] The Long-Term Impact of User Account Control. `http://technet.microsoft.com/en-us/magazine/2007.09.securitywatch.aspx`.

[142] The WebKit Open Source Project. `http://www.webkit.org/`.

[143] Frank Tip. A survey of program slicing techniques. Technical report, CWI, Amsterdam, The Netherlands, 1994.

[144] Top 10 Chrome Browser Add-ons. http://www.pcworld.com/article/185744/top_10_chrome_browser_addons.html.

[145] Trident (layout engine). http://en.wikipedia.org/wiki/Trident_(layout_engine).

[146] UAC Designed To Annoy Users. http://www.crn.com/news/applications-os/207100934/microsoft-exec-uac-designed-to-annoy-users.htm.

[147] Understanding User-Agent Strings. http://msdn.microsoft.com/en-us/library/ms537503(v=vs.85).aspx.

[148] Update: 40 Windows apps contain critical bug, says researcher. http://www.computerworld.com/s/article/9180901/Update_40_Windows_apps_contain_critical_bug_says_researcher.

[149] User Account Control. http://en.wikipedia.org/wiki/User_Account_Control.

[150] UTF-8. http://en.wikipedia.org/wiki/UTF-8.

[151] Bezawada Bruhadeshwar V. Sai Sathyanarayan, Pankaj Kohli. Signature Generation and Detection of Malware Families. In *Proceedings of the 13th Australiasian Conference on Information Security and Privacy*, 2008.

[152] Vulnerabilities in Microsoft Office Could Allow Remote Code Execution. http://www.microsoft.com/technet/security/bulletin/ms10-087.mspx.

[153] H.J. Wang, C. Grier, A. Moshchuk, S.T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[154] Tielei Wang, Tao Wei, Zhiqiang Lin, and Wei Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *In Proceedings of the 16th Network and Distributed System Security Symposium*, 2009.

[155] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, 2008.

[156] WebbIE 3.14. `http://www.webbie.org.uk/`.

[157] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[158] What Goes On Inside Windows 2000: Solving the Mysteries of the Loader. `http://msdn.microsoft.com/en-us/magazine/cc301727.aspx`.

[159] Ollie Whitehouse. GS and ASLR in Windows Vista. In *Black Hat DC*, 2007.

[160] Windows DLL Exploits Boom; Hackers Post Attacks for 40-plus Apps. `http://www.computerworld.com/s/article/9181918/Windows_DLL_exploits_boom_hackers_post_attacks_for_40_plus_apps`.

[161] X86 Calling Conventions. `http://en.wikipedia.org/wiki/X86_calling_conventions`.

[162] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. A brief survey of program slicing. *SIGSOFT Softw. Eng. Notes*, 30(2), 2005.

[163] Ru-Gang Xu, Patrice Godefroid, and Rupak Majumdar. Testing for buffer overflows with length abstraction. In *Proceedings of the 7th International Symposium on Software Testing and Analysis*, 2008.

[164] 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.

[165] You receive a "File Not Found" error message in Excel when you open a file by double-clicking the file name. `http://support.microsoft.com/kb/207574`.

[166] Zero-day Windows bug problem worse than first thought, says expert. http://www.computerworld.com/s/article/9180978/Zero_day_Windows_bug_problem_worse_than_first_thought_says_expert.

[167] Qinghua Zhang and Douglas S. Reeves. MetaAware: Identifying Metamorphic Malware. In *Proceedings of the 23rd Annual Computer Security Applications Conference*, 2007.