REINFORCING THE WEAKEST LINK IN CYBER SECURITY: SECURING SYSTEMS AND SOFTWARE AGAINST ATTACKS TARGETING UNWARY USERS

A Thesis
Presented to
The Academic Faculty

by

Long Lu

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
August 2013

REINFORCING THE WEAKEST LINK IN CYBER SECURITY:
SECURING SYSTEMS AND SOFTWARE AGAINST ATTACKS TARGETING UNWARY USERS

Approved by:

Professor Wenke Lee, Advisor
School of Computer Science
Georgia Institute of Technology

Professor Mustaque Ahamad
School of Computer Science
Georgia Institute of Technology

Professor Christopher Kruegel
Computer Science Department
The University of California, Santa Barbara

Professor Mayur Naik
School of Computer Science
Georgia Institute of Technology

Professor Patrick Traynor
School of Computer Science
Georgia Institute of Technology

Date Approved: 21 June 2013

To my parents,

who lit up my life;

To Meng,

who made it colorful.

ACKNOWLEDGEMENTS

My Ph.D. thesis owes its completion to the support of a number of people. I would like to take this opportunity to acknowledge them.

Professor Wenke Lee has been advising my Ph.D. study since 2008. To this day, I am still thankful for his offer of admission to his research group, which allowed me to embark on the most fruitful and enjoyable educational journey in my life. During the years working with Professor Lee, I received a tremendous amount of guidance and support from him, which are essential to overcoming the research challenges and realizing my own potential. His professional vision, critical thinking, and respect for intellectual freedom have a profound influence on my development of the skills that are indispensable to becoming an independent researcher. I am absolutely honored and grateful to be advised by Professor Lee. I will carry this influence to my career and life in the future.

I would like to thank Professor Mustaque Ahamad, Professor Patrick Traynor, Professor Mayur Naik, and Professor Christopher Kruegel, for willing to serve on my thesis committee. Their insightful comments and suggestions have enlightened me to make significant improvements to this thesis.

The formation of this thesis could not be possible without team work. I have been fortunate enough to collaborate with the following brilliant and collegial team players: Dr. Martim Carbone, Dr. Simon Chung, Dr. Weidong Cui, Professor Wenke Lee, Dr. Zhichun Li, Mr. Kangjie Lu, Professor Roberto Perdisci, Mr. Phillip Porras, Dr. Tielei Wang, Dr. Zhenyu Wu, and Dr. Vinod Yegneswaran. I strongly believe that the collaborative environment created by these people is the single most powerful drive that helped me advance my work.

My Ph.D. experience would not have been the same without my friendly and supportive lab-mates. I especially would like to acknowledge the following who have helped with this work: Manos Antonakakis, Martim Carbone, Brendan Dolan-Gavit, Yeongjin Jang, Andrea Lanzi, Bryan Payne, Kapil Singh, Chengyu Song, Abhinav Srivastava, and Junjie Zhang.

The large-scale experiments and operations involved in this work were supported by the cutting-edge computing infrastructure provided and maintained by the Georgia Tech Information Security Center. I am thankful for the administration team led by Mr. Paul Royal and Mr. Adam Allred. Their immediate and professional responses to my requests have made all the differences.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Unwary computer users are often blamed as the weakest link on the security chain, for unknowingly facilitating incoming cyber attacks and jeopardizing the efforts to secure systems and networks. However, in my opinion, average users should not bear the blame because of their lack of expertise to predict the security consequence of every action they perform, such as browsing a webpage, downloading software to their computers, or installing an application to their mobile devices.

My thesis work aims to *secure software and systems* by reducing or eliminating the chances where users' mere action can unintentionally enable external exploits and attacks. In achieving this goal, I follow two complementary paths: (i) building runtime monitors to identify and interrupt the attack-triggering user actions [58, 59]; (ii) designing offline detectors for the software vulnerabilities that allow for such actions [57, 25]. To maximize the impact, I focus on securing software that either serve the largest number of users (*e.g.* web browsers) or experience the fastest user growth (*e.g.* smartphone apps), despite the platform distinctions.

I have addressed the two dominant attacks through which most malicious software (*a.k.a.* malware) infections happen on the web: drive-by download and rogue websites. `BLADE` [59], an OS kernel extension, infers user intent through OS-level events and prevents the execution of download files that cannot be attributed to any user intent. Operating as a browser extension and identifying malicious post-search redirections, `SURF` [58] protects search engine users from falling into the trap of poisoned search results that lead to fraudulent websites. In the infancy of security problems on mobile devices, I built `Dalysis`, the first comprehensive static program analysis framework

for vetting Android apps in bytecode form. Based on `Dalysis`, `CHEX` detects the component hijacking vulnerability in large volumes of apps [57].

My thesis explores, realizes, and evaluates a new perspective of securing software and system, which limits or avoids the unwanted security consequences caused by unwary users. It shows that, with the proposed approaches, software can be reasonably well protected against attacks targeting its unwary users. The knowledge and insights gained throughout the course of developing the thesis have advanced the community's awareness of the threats and the increasing importance of considering unwary users when designing and securing systems. Each work included in this thesis has yielded at least one practical threat mitigation system. Evaluated by the large-scale real-world experiments, these systems have demonstrated the effectiveness at thwarting the security threats faced by most unwary users today. The threats addressed by this thesis have span multiple computing platforms, such as desktop operating systems, the Web, and smartphone devices, which highlight the broad impact of the thesis.

CHAPTER I

INTRODUCTION

1.1 Thesis Overview

The increasing efforts of securing computer and communication systems has greatly reduced the attack surface of today's software and networks, making extremely difficult to directly launch remote attacks. As a result, cyber attackers are switching to the new strategies that are more subtle and often require victims to unintentionally expose extra attack surface. Thanks to the vast number of unwary and uninformed computer users who are incapable of recognizing their attack-enabling actions, such as visiting a compromised webpage or downloading malicious software, the new attack strategies have seen a wide adoption during the past a few years and helped attackers bypass the-state-of-art defenses, as highlighted by the recent system intrusions into Google and RSA [15, 18], as well as the surge of the so-called watering hole attacks [20].

In face of this new and rising trend of conducting cyber attacks, most existing defense and mitigation techniques become inadequate or even ineffective. This is largely because they focus on voiding the conditions that were necessary to the success of attacks but now are no longer required thanks to the exploitation of unwary users. Traditional firewalls and intrusion detection systems serve as a clear example: their designs unanimously assume that external attacks should always be initiated by the remote side, and thus solely focus their attention on inbound connection requests. However, outbound requests, issued by a misinformed or uninformed user, may also introduce advanced attacks, such as drive-by download, rogue webpages and etc, which easily breaks the aforementioned detections.

In comparison, my thesis work aims to *secure software and systems by reducing or eliminating the chances where users' mere action can unintentionally enable external exploits and attacks.* In achieving this goal, I follow two complementary approaches:

- Building runtime monitors to identify and interrupt the attack-triggering user actions [58, 59];
- Designing offline detectors for the software vulnerabilities that allow for such actions [57, 25].

To pursue a maximum impact, my work focuses on software that either serve the largest number of users (*e.g.* operating systems and web browsers) or experience the fastest user growth (*e.g.* smartphone apps).

Specifically, my work addresses the two dominant attacks through which most malicious software (*a.k.a.* malware) infections happen on the web: drive-by download and rogue websites. `BLADE` [59], an OS kernel extension, infers user intent through OS-level events and prevents the execution of download files that cannot be attributed to any user intent. Operating as a browser extension and identifying malicious post-search redirections, `SURF` [58] protects search engine users from falling into the trap of poisoned search results that lead to fraudulent websites. In the infancy of security problems on mobile devices, I built `Dalysis`, the first comprehensive static program analysis framework for vetting Android apps in bytecode form. Based on `Dalysis`, `CHEX` detects the component hijacking vulnerability in large volumes of apps [57].

1.2 Blocking Web-borne Malware at OS Level

Accounting for the majority malware infections on the web, drive-by download, launched by a booby-trapped webpage, exploits vulnerabilities inside visitors' web browsers, email clients, or even operating systems, in order to surreptitiously install persistent malware. Current preventive efforts are largely passive and attack-specific.

They patch the vulnerabilities and block the offending websites only when new attack intelligence becomes available. As indicated by the recent Google and RSA incidents, such efforts are useless to fresh attacks and often incur a response period that is long enough for attackers to quickly accumulate victims or launch targeted attacks.

Recognizing that drive-by download requires a user to willingly visit a webpage without knowing the consequent attack, I designed and built `BLADE` to robustly prevent web surfers from the entire class of drive-by download, regardless of the targeted vulnerabilities or other attack specifics [59]. `BLADE` alerts the absence of user intent for browsers to download and execute remote files (*i.e.* coerced by an ongoing attack), and in turn blocks the surreptitious install of malware. The research problems, involving multiple aspects of OS managing user interactions, networking, and filesystems, lie in the fundamental tasks of `BLADE`: (i) infer authentic download intent from user interaction events; (ii) correlate inferred user intent to resulting download files; (iii) prevent unintended download files from being loaded as binary code into the memory. `BLADE` relies on hardware interrupt events (*i.e.* unforgeable by software) to infer user intent. `BLADE` listens to the `IOCTL` messages from human input devices to the OS (*e.g.* raw mouse clicks and keyboard strokes) and recovers their semantics by referring to the `GUI` subsystem. Once a download intent is captured, represented by the expected network source and the file information, `BLADE` discovers the resulting download file and validates its origin by matching the file content with recorded network traffic from the source. For files that are downloaded without a direct user intent, consisting of legitimate webpage-included files and dropped malware, `BLADE` makes them only accessible to their respective owner processes and disallows their execution and propagation, so that the normal functioning of browsers remains intact and malware stay harmless. I implemented `BLADE` on Windows as a collection of kernel drivers, compatible with all major web browsers.

`BLADE` has gained a considerable amount of media coverage and technology transfer

inquiries from the industry. It has also been incorporated into a large-scale web security survey project funded by the World Economic Forum. Working on `BLADE` strengthened my expertise and interest in the security aspects of operating systems and web technologies. It enlightened me with an overall picture of the real-world threats plaguing current web users.

1.3 Preventing Rogue Websites at Browser Level

Tricking visitors with sophisticated social engineering techniques, rogue websites have seen a dramatic increase in its share of today's illicit websites, primarily aiding malware propagations and scam promotions. This trend reflects an important shift of cyberattack strategy—from solely targeting software vulnerabilities to a combined approach that also exploits unwary human users. This shift, partly forced by the efforts of making software less exploitable, reveals an alarming fact that current designs of software and security countermeasures have failed to prevent the users from unintentionally or deludedly enabling attacks. My work aims to fill this blank.

Leveraging on browser instrumentation, my work addresses the rogue website prevention via two complementary approaches: (i) check for signs of fraudulence as the current webpage unfolds; (ii) check for signs of approaching a rogue website as the browsing session proceeds. Each approach captures the invariants of different classes of rogue websites. Following the first approach, I designed a method, applying image and text matching techniques on fine-grained `HTML` components, to measure the human-perceptible resemblance among webpages. It is particularly useful for detecting rogue websites that pose as providing well-known services, because these websites bear inherent visual resemblance to the legitimate providers. The prototype I developed is now protecting the users of Microsoft Bing and Yahoo from stumbling upon scareware websites [76].

The second approach are suited for rogue websites that do not exhibit invariants

in their appearances. Rogue websites usually require certain types of referring process to lure visitors and cheat their trust. Among many options, search poisoning has quickly emerged as attackers' favorite choice, maneuvering search engines into displaying arbitrary webpages as part of the search results for popular keywords. To prevent users from falling victim to poisoned search results and the rogue websites hidden behind, I designed and built `SURF` [58], a browser extension that inspects live browsing sessions and alerts the user in realtime when a poisoned search result is encountered. From three sources—internal events of the browser, network level information, and search term characteristics—`SURF` extracts nine detection features, which are carefully selected and evaluated in terms of their individual robustness and collective distinguishability. Based on these features, a decision tree model, trained with large volumes of real world samples, determines if the current browsing session is heading to a rogue website under the disguise of a relevant search result. `SURF` has made two major contributions. First, its realtime, in-browser, and malice-agnostic detection method directly protects web surfers from all kinds of social-engineering-based websites that employ search poisoning. Second, `SURF` prototype allows for the first large-scale empirical study of search poisoning in reality, which spans seven months in time and monitors poisoned search results in major search engines. The result reveals shocking facts that hint at the ineffectiveness of existing detection methods, and it offers insights into the highly stealthy and efficient operations behind search poisoning attacks.

1.4 Vetting Smartphone Apps at Market Level

I have recently expanded my research onto the smartphone platforms with the same focus on protecting end users, as mobile devices continue to gain huge numbers of users and offload more and more user-centric applications from traditional computers.

The design of smartphone OS adopts rather strict policies that govern apps' development and distribution, enforce least-privileges and isolation among apps, and forbid modifications to the underlying system and hardware. Having significantly raised the bottom line of smartphone security, these policies, however, may be put in vein by users who unknowingly install vulnerable or malicious apps. Due to the inadequate app quality assurance and large presence of amateur developers, a significant amount of vulnerable apps may have been produced and found their way to users' devices, as revealed by the Facebook and Skype incidents.

The centralized app distribution model provides an opportunity to vet apps before releasing them to users. To support research along this line, I designed `Dalysis` [57], a comprehensive program analysis framework for Android apps that aims at enabling market-scale vetting tasks. Its front end, built from scratch to consume off-the-shelf Android apps, compiles the Dalvik bytecode into an intermediate representation (`IR`) used by the `IBM WALA` project, and then converts the `IR` into the static single assignment (`SSA`) form. Comparing with decompilation-based approach, Dalysis takes more efforts to implement yet avoid loss of accuracy and unscalable overhead. The extensible back end hosts an array of basic program analyzers, including call graph builder, data-flow analyzer, and *etc.*, which serve as building blocks of vetting methods for different types of vulnerabilities and malice.

Component hijacking is a broad class of vulnerabilities that allow unauthorized apps to access the private or protected resources of the vulnerable app via its public components. I modeled this vulnerability from a data-flow perspective and designed `CHEX` [57], a vetting method that searches for hijack-enabling data-flows on an augmented system dependence graph (`SDG`) derived from the app code. The event-based programing paradigm of Android apps imposes challenges, such as multiple entry points and asynchronous components, to the inter-procedural and context-sensitive analysis. `CHEX` overcomes them by proposing a novel entry point discovery algorithm

and the app-splitting technique to efficiently model the asynchronous executions, which can also serve other types of app analysis. As I did for my other works, I evaluated `CHEX` by conducting a large-scale empirical experiment, which tested over five thousand real apps and confirmed the capability of `CHEX` to accurately vet large volumes of apps. My case studies on the vulnerable apps found during the experiment show that component hijacking can enable a variety of attacks, including those previously unseen on Android platform, such as script injection and data tampering. `CHEX` is now being commercialized by NEC Corp.

In addition to static analysis, my research also involves designing dynamic program analysis methods to investigate apps' impact on users' privacy and security. A very recent project of mine addresses the lack of comprehensive understanding about the causes and approaches for apps to access and handle users' private data, which is necessary for the research community to come up with generically applicable yet sufficiently restrictive policies and mechanisms that govern privacy consumptions on smartphones. To do so, we built a specialized information flow tracking tool, which rewrites `.Net` assembly of Windows Phone apps to conduct instruction-level instrumentation. It not only tracks the acquisition and propagation of private data inside an app as it runs, but also records detailed execution context along the tracked paths, providing semantics of movement and manipulation of private data. We plan to integrate our analysis into the app reviewing process at Windows App Market and reveal the private data consumption based on a large volume of apps.

1.5 Thesis Contributions

In summary, this thesis makes the following technical contributions to the security research community:

1. It proposes, explores, realizes, and evaluates a new perspective of securing software and system, which limits or avoids the unwanted security consequences

caused by unwary users. It also clarifies the importance of considering the presence and impact of unwary users when designing security methods and systems.

2. The knowledge and insights gained throughout the course of developing the thesis have advanced the communitys understanding of several emerging threats and related problem;

3. Each work included in this thesis has yielded at least one practical threat mitigation system, including several that have been adopted in real-world products and services;

CHAPTER II

BLADE: AN ATTACK-AGNOSTIC APPROACH FOR PREVENTING DRIVE-BY MALWARE INFECTIONS

Web-based surreptitious malware infections (i.e., drive-by downloads) have become the primary method used to deliver malicious software onto computers across the Internet. To address this threat, we present a browser-independent operating system kernel extension designed to eliminate drive-by malware installations. The BLADE (Block All Drive-by download Exploits) system [59] asserts that all executable files delivered through browser downloads must result from explicit user consent and transparently redirects every unconsented browser download into a nonexecutable *secure zone* on disk. BLADE thwarts the ability of browser-based exploits to surreptitiously download and execute malicious content by remapping to the filesystem only those browser downloads to which a programmatically inferred *user-consent* is correlated, BLADE provides its protection without explicit knowledge of any exploits and is thus resilient against code obfuscation and zero-day threats that directly contribute to the pervasiveness of today's drive-by malware. We present the design of our BLADE prototype implementation for the Microsoft Windows platform, and report results from an extensive empirical evaluation of its effectiveness on popular browsers. Our evaluation includes multiple versions of IE and Firefox, against 1,934 active malicious URLs, representing a broad spectrum of web-based exploits now plaguing the Internet. BLADE successfully blocked all drive-by malware install attempts with zero false positives and a 3% worst-case performance cost.

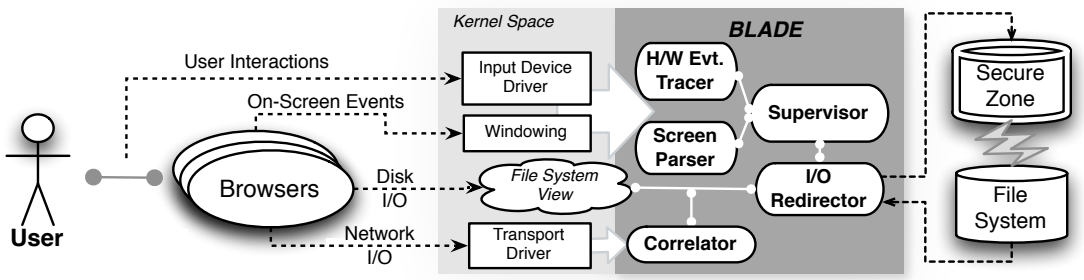


Figure 1: Overview of the BLADE system architecture

2.1 Drive-By Exploit

The web is an increasingly treacherous place. The mere act of connecting one's web browser to the wrong website may result in the installation of an application without the user's authorization or knowledge. Furthermore, attempting to limit one's browsing behavior to reputable and well-known websites is becoming a less effective strategy, as malware developers are actively infiltrating these sites to spread their malicious links [60].

To understand how BLADE defends client browsers from the current generation of drive-by exploits, we first provide a more refined explanation of how drive-by exploits operate. We can then identify the underlying common transaction performed by drive-by exploits that BLADE ultimately aims to stop.

A drive-by download can be described as a series of steps that the adversary performs to achieve the surreptitious download and installation of malware via the victim's browser. The goal of the drive-by exploit is to take effective, temporary control of the client web browser for the purpose of forcing it to fetch, store, and then execute a binary application (e.g., .exe, .dll, .msi, .sys) without revealing to the human user that these actions have taken place. We present the drive-by exploit strategy as a series of phases.

Shellcode injection phase: The first challenge in delivering the drive-by exploit is that of gaining temporary control of the browser. Uniformly, all drive-by exploits begin with a remote code injection, such as buffer overflow exploit against some component within the browser process, e.g., the ActiveX interpreter, a multimedia plug-in, the PDF helper object, the Flash player etc.

Shellcode execution phase: Regardless of which exploit technique is selected by the malware author, the objective of this exploit is to inject a small shellcode segment within the browser process to conduct covert binary installation (this essentially defines the attack as a drive-by exploit).

Covert binary install phase: The final phase of the drive-by exploit is the sequence of steps leading to the final, permanent infection of the client host. Here, the shellcode effectively coerces the now tainted browser into fetching a remote malware application from some remote source on the Internet, storing it within the filesystem and executing it on the victim's host.

2.2 Threat Model, Design Objectives, and Challenges

In our threat model, an adversary conducting drive-by download attacks is allowed to hijack control of a vulnerable browser and inject remote code. BLADE assumes that this attacker should have no persistent malware deployed on the target host in advance, as otherwise the goal of the attack would have been already achieved. Specifically, there is no rootkit from the adversary installed on the system, i.e., the OS kernel is trusted. Although scenarios where the assumed attacker can remotely exploit a kernel vulnerability via a browser exist, which are out of the scope of this model, we argue that they are extremely rare and could be addressed by integrating orthogonal OS integrity protection technologies, such as hypervisor-based protections, with BLADE.

BLADE does *not* attempt to halt the drive-by exploit at the shellcode injection

phase or the shellcode execution phase. Given the overwhelming diversity of browser extensions, modules, and code changes that are continually churned out by the high-paced browser development community, the task of stopping all shellcode injections is a truly daunting challenge. Even with the introduction of OS-level protections such as DEP and ASLR and browser-level sand-boxing, drive-by exploits are still succeeding [47].

Rather, BLADE incorporates a different tactic in fighting drive-by attacks. From BLADE's perspective, the drive-by download attack conducts a series of steps designed to bypass the normal user-content-handling procedure that should be performed whenever a browser attempts to store this data to disk. The fetched binary itself represents an unsupported browser type that cannot be handled and rendered directly by the browser, but must be delivered through the standard user-initiated consent-to-download dialog. BLADE aims to disrupt the covert binary install phase, completely agnostic of which browser component was exploited or which shellcode injection strategy was employed to achieve the initial browser hijack.

BLADE's core mission is to foil the execution by any program entity (including the OS), of any on-disk data content received through the browser process tree, unless that content can be correlated with a user consent dialog event. BLADE enforces this requirement while not interfering with normal browser operations in any way. Specifically, we can accommodate automated software updating that is a common practice among browsers and their plug-ins through source domain whitelisting [1]. Browser native code execution mechanisms (*e.g.* Native Client) are not affected by BLADE since they rely on the preinstalled client, rather than the OS, to load and execute the code.

Inherent in this task are several key technical challenges, which we outline here and further cast as design goals that we directly address in this paper:

[1] Our current prototype does not implement this capability.

• *Real-time user authorization capture and interpretation* – BLADE must monitor user-to-browser interaction events to capture explicit user authorizations that permit upcoming download actions. From each captured authorization, BLADE must extract identity information pertaining to the expected download (*i.e.* remote URL, file name, and local path) in order to uniquely identify the resulting file.

• *Robust correlation between authorization and download content* – BLADE must programmatically distinguish user-initiated browser downloads from unauthorized ones and reliably correlate every authorization event with the corresponding binary stream that is downloaded by the browser from the network.

• *Stringent enforcement of execution prevention* – Files containing unauthorized download content must be stringently prevented from execution, while other types of access from supervised processes are allowed. This enforcement must not impede normal operations of browsers as well as other programs.

• *Browser agnostic enforcement* – BLADE must not depend on either the integrity of browsers or their internal handling of tasks. We must assume that new browser attack strategies will continue to evolve along with the rapid development of new browser technologies. Browser updates or potential browser compromises caused by inevitable software vulnerabilities must not affect the protection quality BLADE provides.

• *Exploit and evasion independence* – BLADE's enforcement mechanism must be entirely agnostic to exploits employed as the first step to subvert the browser into performing drive-by downloads, and thus be immune to all kinds of sophisticated evasion techniques including code obfuscations and zero-day vulnerabilities.

• *Efficient and usable system performance* – BLADE's performance impact on browser content handling must be negligible. Overall, BLADE should not impose perceptible delays to normal browser operations, and have no impact on non-browser host operations.

Considering the threat model and design trade-offs, we believe that placing BLADE as a dynamic loadable driver into the OS is a viable design choice to achieve our goals listed above. To reliably capture and interpret user interactions and guarantee unforgeability, BLADE has to reside at least as low as the OS. Even in scenarios where virtual machine monitoring systems are deployed, having BLADE inside the kernel is more efficient than placing BLADE-equivalent functionalities inside the hypervisor and more accurate than solely using virtual machine introspection.

2.3 BLADE System Architecture

Figure 5 illustrates the BLADE software architecture and its core components. The front-end components, including the Screen Parser, Hardware-Event Tracer and Supervisor are responsible for collecting information displayed on the screen and tracking user interactions when necessary. The Screen Parser monitors kernel windowing events as the status of on-screen UI changes in real time. It signals the Supervisor upon the appearance of a *download consent dialog* (or authorization dialog) on the screen foreground and reports necessary information parsed from the screen (see § 2.3.1). A download consent dialog is defined as any prompt (dialog box) created by browsers or plug-ins seeking download permission from the user. Due to the well-defined application interface used by commodity browsers to implement download confirmation dialogs, a small number of signatures (one or two per browser family) are needed to capture all download consent events. Each signature captures the external appearance and the internal hierarchy shared by all UI instances of that class. The Screen Parser uses these signatures to discover download consent dialogs, locate the respective positions of confirmation elements on these UI dialogs (*e.g.* the "Save" button), and extract the download identity information (*e.g.* URL, file name) to be used in the correlation process. Upon receiving the signal from the Screen Parser, the Supervisor invokes the Hardware-Event Tracer to intercept subsequent mouse

and keyboard input events that would trigger the download confirmation. BLADE relies on hardware events as the only dependable source of extracting user consent information due to their unforgeability in our threat model (see § 2.3.3).

The Correlator and the I/O Redirector form the back end of the BLADE system. They correlate inferred authorizations from the front end with resulting downloads and enforce the nonexecution policy for downloads that are not directly requested by the user. The Correlator ensures BLADE's resistance to spoofing attacks such as forged UI dialogs (discussed in § 2.4), by virtue of its capability to validate the authenticity of the consequent file corresponding to a user download consent. We define the *download identity information* as (*URL, Path*), i.e., a 2-tuple of the remote URL and the local storage path, to uniquely delegate a user download authorization. The Correlator matches a file f with a tuple (u, p) when f is saved at p with data content received from u (see § 2.3.4).

The I/O Redirector persistently guarantees that uncorrelated downloads can do no harm by establishing the *secure zone*. As its name suggests, the I/O Redirector intercepts disk write operations initiated from the browser process tree (namely, *supervised processes*) and redirects them to the secure zone, where execution is explicitly prohibited by blocking *memory-section synchronizations*. We describe this in more detail in § 2.3.5. By default all files downloaded by supervised processes are transparently redirected to the secure zone. Files that pass the download correlation process (i.e., where the content written is indeed from the user-authorized remote URL) are subsequently moved out of the secure zone back to their original destination in the file system. This move is accomplished by modifying file system metadata as opposed to copying the downloaded data, which can be finished in constant time. Our design of the secure-zone-based I/O redirection with the capability to discern user-initiated downloads enables a generic defense strategy that targets the common behavioral pattern shared by all drive-by download attacks.

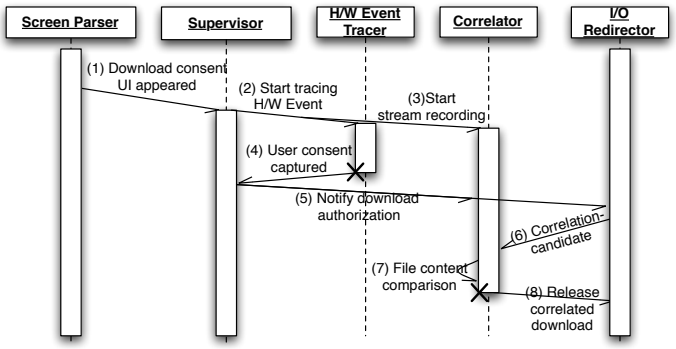


Figure 2: Download authorization workflow

We now discuss the design details of the BLADE architecture components, in the order of web download and authorization workflow as shown in Figure 2.

2.3.1 Screen Parser

BLADE's download authorization lifecycle is triggered by the appearance of download consent dialogs, which seek user's permission on downloads. Internally, every status change of UI elements causes a certain windowing event to be sent to the operating system, which express the change by re-drawing the screen. For instance, creating a new window causes an `OBJ_CREATE` event to be generated on Windows platforms, which contains information needed by the operating system to draw the new window on the screen (*e.g.* position, size, text). The Screen Parser component of BLADE relies on accurate interpretations of these windowing events intercepted from within the OS to discover download consent UI elements and effectively monitor content displayed on the screen.

Since significant performance degradation can be introduced if suboptimal methods are employed, this component merits considerable care in implementation. For example, a naive option to implement the Screen Parser is as a direct hook into windowing event handlers. However, such implementation would block the window drawing process while trying to recognize newly visible UI elements, and in turn,

result in perceptible UI delays when the window being parsed contains too many elements.

To optimize performance, BLADE implants an *agent* in user space to prefilter irrelevant windowing events. It runs in parallel with the window management routines, asynchronously filtering and preparsing windowing events in the user space that would otherwise incur significant kernel CPU cycles if directly handled by the Screen Parser. The agent pipes its output to the Screen Parser, which *may* represent a user consent dialog currently in focus. To secure against interference from untrusted user-level programs, an independent sanity checker in the Screen Parser cross-validates the input from the agent by inspecting kernel memory objects representing the UI elements.

On the Windows platforms, handling only three types of events is sufficient to completely cover the real-time changes of the currently focused window: `EVENT_SYSTEM_FOREGROUND`, `EVENT_SYSTEM_MOVESIZEEND`, and `EVENT_SYSTEM_MINIMIZEEND`. Key strokes triggering a particular UI element can also be obtained as one of the associated attributes. Screen information is parsed only if the newly focused window is deemed to represent a request for download permission.

UI signatures are used to identify download consent dialogs and guide information extraction from these dialogs. Each signature describing the internal composition shared across all UI instances of a class is sufficiently general and accurate in capturing all dialogs with the same look and feel as the sample used for signature generation. Due to the uniform use of interfaces by current browsers to request download permissions, there are only a handful of UI classes that serve this purpose, which also remain highly stable across browser versions and regular updates. Hence, using only two signatures for Firefox and one signature for IE, we can successfully capture all forms of download notifications in these browser families across versions.

Note that attempted evasions by faking user consent dialogs may trigger a signature match, but cannot elude the Correlator (see § 2.3.4 for the correlation process).

2.3.2 Supervisor

As the first component loaded upon BLADE startup, the Supervisor serves the role of coordinator for carrying out all tasks of BLADE. It is charged with assigning tasks to other BLADE components and coordinating their execution, as responding to the different event notifications from the Screen Parser. The Supervisor also takes care of internal communications among all BLADE components, including user-kernel communication backed by IOCTLs (device input and output control), and kernel-kernel communication implemented by simply sharing a nonpaged pool across all kernel components as a means of information exchange. Here, spin-lock-based synchronizations are used to protect the integrity of shared data.

Upon notification of the appearance of a download consent dialog, or a status change to an existing one, the Supervisor initiates other kernel components accordingly, or resets them in response to status changes. As shown in Figure 2, when a new relevant UI element is discovered, the Hardware Event Tracer (H/W Event Tracer) is triggered, with input information such as the on-screen locations of download consent dialogs. Its task is to sense the user-invoked hardware device signals that may indicate the user's consent to permit a pending download request. The Correlator also receives a command from the Supervisor, indicating that the corresponding stream recording process should start. A download authorization is not recognized by the Supervisor until user consent is captured by the H/W Event Tracer (in the form of physical mouse clicks or keystrokes).

The Supervisor also actively maintains a complete list of supervised processes, on which most BLADE routines rely to function correctly. For example, the I/O Redirector and the Correlator only intercept file operations and record inbound network

streams of supervised processes. The list is initialized to be empty when BLADE starts. A process p will be added into the list when (a) it is a newly created browser process, (b) it is a newly created process spawned by a supervised process, or (c) a remote thread is created within the process by a supervised process. Tracking remote thread creations is critical for blocking I/O redirection evasions, which may employ a remote thread to carry out disk I/O on behalf of an unsupervised process. The consequent list of this logic covers all possible execution entities that might either initiate a legitimate browser download or be exploited to deliver surreptitious downloads. Listed processes will be removed as they are terminated. The Supervisor registers a callback routine for process creation and termination events by calling `PsSetCreateProcessNotifyRoutine`.

2.3.3 Hardware Event Tracer

Once a download consent dialog is identified by the Screen Parser, the next task is to interpret the user's response. We developed the Hardware Event Tracer (HET) to track user interactions with this UI element by monitoring signals generated from the hardware to the OS. Signals at this level can never be forged by attackers in our threat model; thus, BLADE is immune to attempted evasions by faking an affirmative response to user download consent events.

The HET starts with a notification from the Supervisor indicating the appearance of a certain download confirmation UI. The HET's role is to capture responses from the user's mouse clicks or keystrokes. During the tracing interval, which normally lasts a few seconds, the HET looks for any mouse click whose on-screen coordinates fall in the areas of download consent dialogs, and any keystroke that can trigger these UIs. The HET also maintains some state information in order to make accurate decisions regarding whether the intercepted hardware events could finally trigger the download consent. The HET terminates the tracing activity due to status changes

from the on-screen consent dialog (*e.g.* minimized, unfocused).

Our current prototype implements the tracking routines only for pointer input devices, which means that users can express their consent only by using the mouse. However, adding support for keyboard input using the same principle should be straightforward. Moreover, the performance overhead introduced by the addition of keyboard tracing is expected to be minimal simply because keyboard events are less frequent than mouse events in web browsing.

2.3.4 Correlator

One of the key challenges in BLADE is establishing the 1-1 mapping between user download authorizations and downloaded files. The Correlator addresses this problem and ensures the authenticity of user-consented file downloads. Guaranteeing authenticity prevents potential attacks seeking to deliver a malicious download, either by prompting deceptive dialogs or subverting benign browser downloads.

Since BLADE is independent of the browser and treats it as a black box, only the external behavior of the browser (*e.g.* interactions with OS) is visible to it. Hence, the Correlator analyzes information available in the OS kernel, oblivious to the internal download handling of browsers. As browsers invariably rely on the OS to provide network and file system capabilities, all kernel drivers including the Correlator have the chance to peek into each transaction and retrieve a wealth of information about it. For example, network traffic incurred by a browser is fully transparent to the Correlator (at multiple kernel system levels) while it is being processed in the OS network protocol stack. Similarly, the Correlator can intercept file system access operations from the browser.

The Correlator associates a file download with a user authorization in two steps: it discovers the *correlation candidate* file, and then validates its authenticity. We now demonstrate why a file that passes these two steps while being correlated with a given

user authorization is assured to be in compliance with that authorization.

Recall the tuple form of an inferred user authorization discussed earlier in this section (*URL, Path*). Here, we use the second element, the destination path in the local storage, plus the file name as a criteria for discovering the correlation-candidate. Whenever a file has been written with the same path and name as that of a pending authorization, the Correlator marks it as a correlation candidate and starts the validation process immediately. We call it a candidate because the adversary in our threat model is able to replace the file content after fully compromising the browser.

The first element of the authorization tuple, the source domain, indicates the origin location of the file content and is used for source validation. We implement the following source validation technique based on content comparison. First, we keep a log of inbound transport-level stream for each TCP session created by supervised processes, which is later compared with the download candidate. If the content of a particular download-candidate appears in a stream log that corresponds to the source URL recorded in the authorization tuple, the candidate is validated and the correlation process completes with the candidate being correlated with the user authorization. Our content-comparison approach works even when encryption is used (*e.g.* HTTPS, VPN), because browsers rely on OS support to process transactions of this kind. The user-level APIs are simply wrappers for kernel functions and therefore plaintext content can always be obtained by kernel drivers prior to encryption (when sending) or after decryption (when receiving). Furthermore, browser-level compression/encoding schemes (*e.g.* SDCH), which only apply to web streams that are natively rendered by browsers, do not interfere with the BLADE correlation process.

Although the source validation idea is straightforward, an efficient comparison algorithm and a reliable implementation require careful consideration. From a performance perspective, the Correlator should avoid unnecessary stream logging and

halt ongoing log writes once they are deemed unnecessary. Moreover, stream recording needs to be performed only when there is an incoming authorized download file and needs to consider only inbound content. Hence, a new logging process will be initiated, only when a download consent dialog requesting a download permission pops up, and only on streams sharing the same remote endpoint as the authorization dialog. A subtle issue is that the source of the download file is identified by a URL on the authorization dialog, while the remote end of a stream is identified by an IP address. The Correlator performs a domain name lookup in the local DNS cache to resolve the corresponding IP address(es). Integrity of the local DNS cache is guaranteed in our threat model because it is being maintained by a trusted kernel component. The logging terminates either when the user denies the download request or after the last stacked authorization permitting that source has been correlated with a file download. As native browser downloaders are all single-threaded, our current prototype does not support the case of multi-threaded downloads where content of a single file comes from multiple streams.

2.3.5 I/O Redirector

BLADE introduces the *secure zone* (i.e., a virtual storage area) as a mechanism to restrict execution of disk footprints that are caused by supervised processes but not explicitly allowed by users. Unlike sandboxing, which blindly isolates execution of untrusted code, the secure zone of BLADE is intelligent enough to *selectively contain* potential threats and ultimately prevent them.

The design philosophy of the secure zone is based on the *closure property* of browser disk writes derived from our study of generic disk access patterns by browsers. The robustness of this property was evaluated by exercising popular browsers with multiple web browsing workloads (see § 2.6.3).

Closure property: On a clean computer running commodity OS and browsers,

let $P = \{p \mid p : \textit{any browser process}\}$, and F, F_{auth}, F_{int} and F' be four sets of files on disk:

$$F = \{f \mid f : \textit{ any file written by } p, \textit{ where } p \in P\};$$

$$F_{auth} = \{f_a \mid f_a : \textit{ any -authorized browser download}\};$$

$$F_{int} = F - F_{auth} \quad (\textit{given } F_{auth} \subset F \textit{ is always true});$$

$$F' = \{f' \mid f' : \textit{ any file opened by } p', \textit{ where } p' \in \bar{P}\};$$

We observe that $F_{int} \cap F' \approx \emptyset$. This implies that, except for user-authorized download files (F_{auth}), any other file to which browser process (P) writes data is not normally accessed by non-browser processes ($\bar{P}$). More generally, it indicates that the disk data that is written by browsers without explicit user consent is well-contained within an implicit scope on disk, and thus should not be accessed by other processes or executed by any program entity. Discovering this scope inspired our design of the secure zone. The I/O Redirector plays a central role in managing the secure zone by enforcing the following policies:

P1 : Any new file created by a supervised process is redirected to the secure zone.

P2 : Any existing file modified by a supervised process is saved as a shadow copy in the secure zone, without change to the original file.

P3 : I/O redirection is transparent to supervised processes.

P4 : I/O redirection only applies to supervised processes. Files in the secure zone can only be accessed via redirection.

P5 : No execution is allowed for files in the secure zone.

P6 : Any file correlated with a user download authorization is remapped to the filesystem.

Together these policies enable a complete containment of disk footprints affected by content delivered through browser processes without user knowledge, while still preserving the browser usability due to transparency.

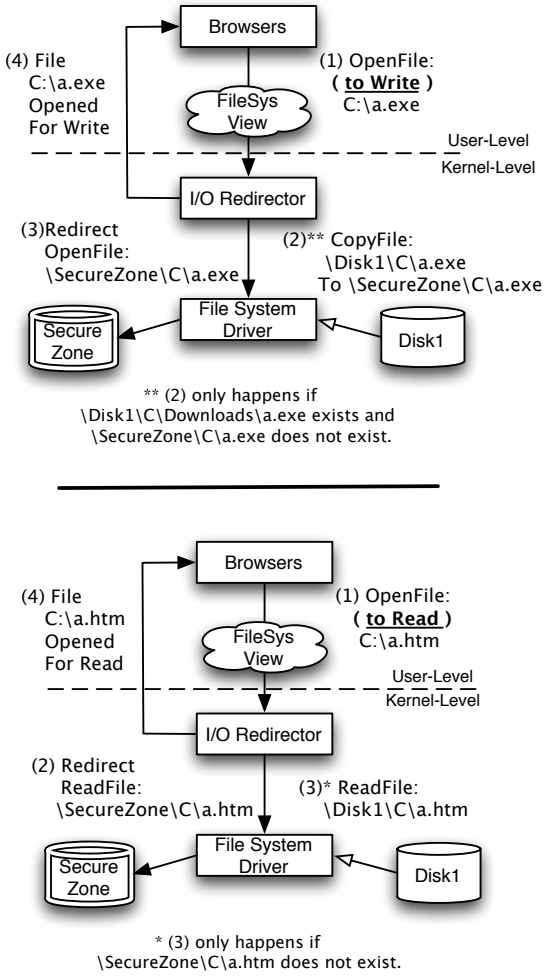


Figure 3: Browser file access requests processing by the I/O Redirector (top: write; bottom: read)

Figure 3 provides a high-level overview of how the I/O Redirector handles the two types of file accesses in order to enforce $P1$ – $P3$ listed above: the upper subfigure shows that the browser is trying to *write* `C:\a.exe` to the disk (*i.e.*opening a file handle with write privilege). Upon receiving the request, the I/O Redirector first checks the existence of the file's shadow copy "`\SecureZone\C\a.exe`". If it exists, *i.e.*the file has been previously created or modified by the supervised process, the I/O Redirector immediately forwards the request down to the file system driver with the target being modified into the path of the shadow copy. Otherwise, the I/O Redirector might need to create such a shadow copy before modifying and redirecting

the request, depending on whether the request is to create a new file "`Disk1\C\a.exe`" or to modify/replace an old one. Finally, the browser that obtains the returned file handle is unaware that it is operating on a shadow copy of the file in the secure zone. The lower subfigure shows that a *read* request is redirected to the shadow copy "`\SecureZone\C\a.htm`" if it exists. Otherwise, the request is passed down to the file system without the need for redirection. The I/O Redirector also provides a different file system view to supervised processes, which hides the separation of files inside and outside the secure zone.

To enforce $P4$, which guarantees the nonpropagation property of files in the secure zone, the I/O Redirector simply passes through file access requests from processes that are not supervised (*i.e.*no redirection happens), except for denying those that are obtaining handles to files in the secure zone. The policy $P5$, file execution prevention, is performed by blocking executable images from being mapped into the memory. Specifically, the I/O Redirector intercepts `AcquireForSectionSynchronization` operations on files located in the secure zone. This is a necessary operation performed by the Windows kernel to load all forms of executables including normal program (.exe, .msi) startups, dynamic library (.dll) loads and driver module (.sys) installations.

When the Correlator successfully matches a previously inferred user download authorization with a file written to the secure zone, the I/O Redirector is notified and the file is remapped back to the filesystem instantly.

2.4 Security Analysis

In this section, we analyze the soundness of the BLADE system design by discussing various attacks that knowledgeable adversaries may pursue to circumvent BLADE. For each attack strategy, we identify the countermeasures that have been incorporated into the BLADE design to address these threats.

Spoofing attacks – The attacker may attempt to drop malware directly onto the

local file system, without being redirected to the secure zone. To accomplish this, the attacker must (a) fool BLADE by forging a fake download consent dialog and the user response, or (b) fool user and the BLADE Screen Parser by spoofing browser GUI to display rogue download confirmations [27]. **Countermeasures:** We address (a) by ensuring that the user authorization inference is based on real hardware events that cannot be spoofed at the application layer. The BLADE Correlator eliminates the possibility of (b) by taking additional steps to validate the origins of user consented downloads. Although the attacker can launch a denial-of-service attack by disabling the user-level Screen Parser, this action will not lead to the surreptitious infection of the browser's host.

Download injection and process hijacking attacks – The attacker may attempt to move a downloaded malware instance out of the secure zone and then execute it. Having control of the browser process, the attacker could replace an authorized download with malware. The attacker may also hijack an unsupervised process whose file I/O is not redirected, e.g., by creating a remote thread within an unsupervised process. **Countermeasures:** Content-based correlation guarantees the source of authorized downloads and prevents download injection attacks. The BLADE Supervisor follows process creations (i.e., child processes of the browser) and remote thread activations (sibling) to ensure that unauthorized file writes from the supervised processes are appropriately redirected to the secure zone.

Coercing attacks – The attacker may attempt to coerce the operating system to execute the malware directly from the secure zone. **Countermeasure:** Since I/O redirection is implemented and enforced by a kernel driver, we believe that such attacks should be infeasible by design. If one asserts that the malware publisher may have control logic buried within the kernel to halt BLADE's operation, then we argue that the drive-by download attack is entirely unnecessary.

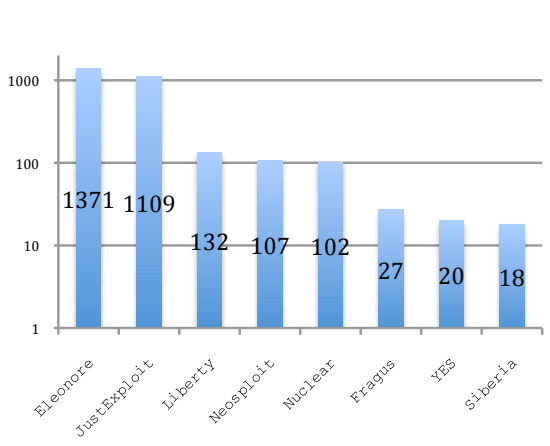
2.5 Limitations

While we believe BLADE represents an effective service for stopping surreptitious drive-by installations of malware, we recognize that it does not provide complete coverage of all threats web users are facing. First, BLADE does not prevent social engineering attacks where the user authorizes the download and installation of malicious binaries disguised as benign applications. Second, BLADE does not prevent in-memory execution of transient malware, which could be scripts such as JavaScript bots or x86 code inserted into memory by exploits. While such attacks are out of scope for our system, the latter attacks could be prevented by orthogonal protection techniques, such as DEP. Third, BLADE is dependent on explicit download-consent UI, which is optional (can be disabled by the user) in certain browsers. Users wishing to use BLADE to protect their web surfing activities must enable download confirmations on their browsers. Finally, BLADE is effective only against binary executables and does not prevent the download and installation of interpreted scripts. However, the overwhelming majority of current drive-by download malware is delivered as binaries. At a minimum, our system raises the bar by rendering the prevalent drive-by download threat obsolete. We intend to explore ways to stop the installation of malicious scripts in the future.

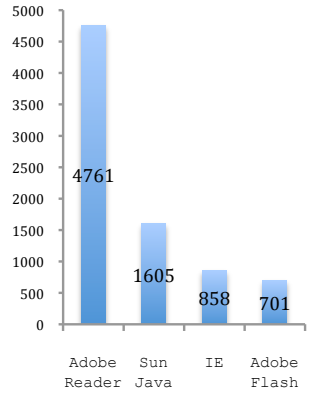
2.6 Evaluation

2.6.1 Empirical Daily Evaluation on Malware URL Lists

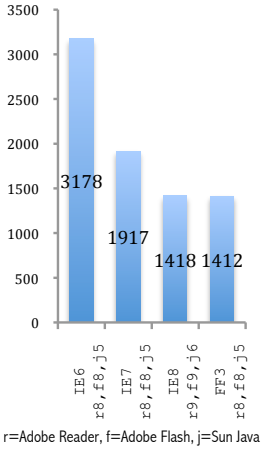
One way to demonstrate the effectiveness of a security system is to exercise it against contemporary real-world threats. Our testbed automatically harvests malware URLs from multiple whitehat mailing lists on a daily basis and evaluates BLADE against potential drive-by URLs that were reported in the past 48 hours. To validate BLADE's browser and exploit independence, each URL is tested against multiple software configurations covering different browser versions and common plug-ins.



(a) Encountered Exploit-Kit Distribution



(b) Vulnerability Distribution



(c) Attack Distribution

Figure 4: Statistics from daily malicious URL experiment

The experiment lasted for 3 months and visited 3,992 unique malicious URLs. The dataset that was collected also offers a glimpse into the Internet's contemporary drive-by malware landscape as summarized in Figure 4. Figure 4 (a) shows the distribution of exploit kits encountered during our experiments illustrating the growing popularity of commercialized exploit kits. Eleonore and JustExploit seem to be the most popular. Figure 4 (b) shows the distribution of attacks by vulnerable software. We find that (*i*) pdf exploits currently dominate, (*ii*) attackers increasingly prefer targeting plug-ins over browsers because of the wider attack surface, and (*iii*) they largely rely on

commercialized exploit kits to launch reliable attacks.

Figure 4 (c) displays the distribution of successful attacks based on browser and software configuration. Not surprisingly, we find that all tested browsers are vulnerable. We find that the Internet Explorer 6.0 system configured with Adobe Reader 8.0, Adobe Flash 8.0, and JVM 5.0 is the most vulnerable. A similarly configured Firefox 3.0 system experiences less than half the number of exploits and is comparable to Internet Explorer 8 running Adobe Reader 9.0 and JVM 6.0. Table 1 summarizes results from our daily evaluation. The number of trials is more than the number of unique URLs because each URL might appear on the list for multiple days and is tested on multiple VM configurations. As shown in Table 1(a), BLADE was successful at blocking all 7,925 attempted drive-by malware installs while generating zero false alarms. Furthermore, all downloaded malicious binaries were safely quarantined into the secure zone. While these results might be surprising at first glance, they are expected because BLADE is designed in an exploit oblivious manner. It is worth noting that at no point did our system design or implementation necessitate additional tuning to handle a new exploit or shellcode type. Table 1(b) provides a summary of the malware binaries captured. The 7,925 trials pushed 9,745 binaries (certain sites push more than one binary) which included 8,126 EXEs and 1,619 DLLs. The average detection rate of these binaries from `virustotal.com` was only 28.43%.

Only about half of the malicious URLs were observed to be delivering drive-by download attacks when tested. While we do not know the exact reason why attacks fail in each instance, they include the following: (a) malicious sites that have been cleaned up, (b) misclassified sites (e.g., phishing sites) that do not attempt surreptitious drive-by downloads, (c) sites that employ IP tracking to blacklist repeated visitors, and (d) sites that target vulnerabilities not present in our configuration.

(a) Evaluation Metrics

	Total	True Positive	False Positive	True Negative	False Negative
Trials	18896	7925	0	10971	0
URLs	3992	1934	0	2058	0

(b) Dropped File Statistics

Malware Intercepted	File Types		AV coverage Day-0 (VT rate)
	EXE	DLL	
9745	8126	1619	28.43%

Table 1: Results from daily malicious URL experiment

2.6.2 In Situ Attack Coverage Evaluation

The first experiment demonstrates BLADE's effectiveness against thousands of drive-by download attacks in the wild. However, it is possible that attacks in the wild are dominated by a few exploit kits and exercise only a limited set of common exploits. To compensate for this potential limitation, we conducted a second experiment that specifically evaluates BLADE against a wider set of hand-crafted attacks and more browser versions. Specifically, this customized attack set is composed of diverse shellcodes and exploits targeting several vulnerabilities in browser/plug-in software including 11 recently disclosed zero-day exploits listed in Table 2. In each case, BLADE successfully prevented the execution of the drive-by exploit binary, reaffirming our design premise that BLADE delivers complete and accurate protection in a browser-agnostic and exploit-oblivious manner.

2.6.3 Benign Website Evaluation

We evaluate BLADE's effectiveness on benign web sites, i.e., the false positive rate. For BLADE, a false positive implies that the execution of a legitimate (authorized) executable download is blocked by BLADE. Under BLADE's design, there are two potential reasons why an authorized executable download may be inadvertently hindered by BLADE: (*i*) the user's authorization cannot be inferred, which leaves the resulting download in the secure zone as untrusted; (*ii*) a legitimate browser download seeks to execute benign logic without the user's consent, which represents a violation of our root assumption. Thus, we tried to create a workload that might trigger (*i*) or

ID	Exploit CVE-ID	Browser	Exploit Payload	Detected By Blade	Vuln. Notes
1	2006-3677	Firefox 1.5	Remote_shell_bind	**YES**	window.navigator
2	2005-1476	Firefox 1.5	Download_exec	**YES**	InstallTrigger.install()
3	2007-0038	Firefox 2.0	Download_exec	**YES**	LoadAniIcon()
			Dll_injection	**YES**	
4	2009-2477	Firefox 3.5	Download_exec	**YES**	**TraceMonkey 0-day**
			Dll_injection	**YES**	
5	N/A	Firefox 3.5	Download_exec	**YES**	**Wings3D 0-day**
6	2009-0927	Firefox 3.5	Remote_shell_bind	**YES**	Collab.getIcon()
			Dll_injection	**YES**	
7	2010-1028	Firefox 3.5	Download_exec	**YES**	**Font Overflow 0-day**
8	2010-1082	Firefox 3.5	Download_exec	**YES**	**XML Overflow 0-day**
9	2007-0038	IE 6.0.2900	Download_exec	**YES**	LoadAniIcon()
			Dll_injection	**YES**	
10	2006-4777	IE 6.0.2900	Download_exec	**YES**	KeyFrame()
11	2006-3730	IE 6.0.2900	Dll_injection	**YES**	setSlice()
12	2009-0075	IE 7.0.5730	Remote_shell_bind	**YES**	CFunctionPointer
13	2009-1539	IE 6.0.2900	Download_exec	**YES**	**DirectShow/ quartz 0-day**
14	2009-3672	IE 7.0.5730	Remote_shell_bind	**YES**	**IE Style-object 0-day**
			Download_exec	**YES**	
15	2008-0015	IE 7.0.5730	Download_exec	**YES**	**DirectShow/ATL 0-day**
16	2010-0249	IE 6.0.2900	Download_exec	**YES**	**IE Aurora 0-day**
17	2010-0886	IE 7.0.5730	Download_exec	**YES**	**JDK/JRE 6 0-day**
18	2010-0806	IE 7.0.5730	Download_exec	**YES**	**IEpeers 0-day**
19	2009-4324	IE 8.0.7600	Download_exec	**YES**	**PDF/ newplayer() 0-day**
			DLL_inject	**YES**	
			Remote_shell_bind	**YES**	

Table 2: Test results on targeted attacks and 0-days

(ii).

To address (i), we first tested the signature coverage of download consent dialogs for each browser by looking for an unknown method for requesting download consent. We downloaded 30 different software applications from 15 highly ranked freeware sites, with varying file types (.exe, .zip, .msi etc.). We also checked whether download consent UIs can be reliably discovered when noise is introduced onto the screen. Neither of the above two test cases revealed any false positives. We used a stress-testing-based strategy to create a workload that could lead to false positives incurred from (ii). By manually visiting a URL pool, including the top 5 highly ranked websites from 16 categories [1], we verified that BLADE does not disrupt normal browser interactions with these benign sites.

2.7 Related Work

We discuss prior measurement studies that inform the design of BLADE and distinguish it from existing URL analysis services, malware defense systems, and browser-based protections.

2.7.1 Internet Measurement Studies

The problem of drive-by downloads, particularly those resulting in malware installations on unsuspecting victims, has attracted considerable attention from researchers. In 2005, Moshchuk et al. [62] studied the threats, distribution, and evolution of spyware through an examination of more than 18 million URLs, finding scripted drive-by downloads in 5.9% of the pages visited. Seifert et al. [75] examined the prevalence and distribution of malicious web servers using the Capture-HPC client honeypot, identifying more than 300 malicious sites. In [71], Provos et al. provided a detailed dissection of the sophisticated methodology employed by the blackhats and the steps involved in executing a typical drive-by download exploit of a system. In a subsequent study [70], the authors provided extensive quantitative measurements of the global prevalence and distribution of the parties (landing sites, redirection sites and script hosting sites) involved in drive-by downloads by examining billions of URLs in the Google web archive. These studies underscore the significance of the drive-by download malware problem and motivate development of the BLADE system.

2.7.2 Website Survey Systems and Proxy Services

Blacklist services such as [12] provide alerts on malicious software systems and websites, currently listing more than 392,000 malicious sites. Strider HoneyMonkey [85] and phoneyc [65] crawl the Internet looking for websites that host malicious code. While the former approach uses Virtual Machines running different operating systems and patch levels, the latter is a lightweight low-interaction system that emulates

browser execution of JavaScript.

SpyBye [69] operates as a proxy server and uses simple rules to classify a URL into three categories: harmless, unknown, or dangerous. The classification process can be error prone and is meant to be a tool for webmasters to track the security for sites that they administer. The SecureBrowing software plug-in developed by Finjan [11] scans web pages in real time for viruses and malware. While the details of their detection methodology are proprietary, it is presumed to be a combination of attack signatures and URL blacklists. The BrowserShield [74] proxy system uses script rewriting and vulnerability-driven filtering to transform inbound web pages into safe equivalents by disabling execution of malicious JavaScript and VBScript exploits at runtime. Wepawet is an online submission service for detecting and analyzing malicious URLs with the capability of analyzing exploits in Flash, JavaScript, and PDF files [40]. Unlike these approaches, BLADE does not require attack signatures and is effective against zero-day attacks.

SpyProxy [61] is an execution-based malware detection proxy system, that executes active web content in a virtual machine environment before it reaches the browser. A limitation is that protection is guaranteed only when the host machine and the proxy machine maintain the identical software configuration.

2.7.3 Network- and Host-based Malware Defense Systems

Systems such as BotHunter [42] and BotSniffer [43] are meant to detect infected enterprise systems based on post-infection network dialog, but do not prevent the execution of malware. AntiVirus systems [13] and services like CloudAV [67], which attempt to block the execution of malware, are limited by the reliance on binary signatures. For drive-by attacks, BLADE addresses the limitations of these approaches, i.e., it acts like an IPS that thwarts the execution of malware and does not rely on signatures.

Egele et al. [32] proposed the use of x86 emulation techniques to defend browsers against a specific type of drive-by download attack, i.e., heap-spraying code injection attacks. Their objective is similar to that of NOZZLE [73], which uses static analysis of objects in the heap to detect heap-spraying attacks. BLADE differs from these systems in that it does not detect the attack, but rather prevents the execution of the malware. Our approach has the benefit that it defends against all forms of web-based surreptitious-download exploits, including malware installed using heap-spraying code injection attacks.

2.7.4 Sandboxing/Isolation Systems

Solitude [51] and Alcatraz [54] are two systems that limit the effects of attacks by providing support for application-level isolation recovery. Secure browsers [23, 83] have been developed that use sandbox techniques to prevent malware installations. The Chromium sandbox [23] attempts to mitigate browser exploits by separating the trusted browser kernel (which runs with high-privilege outside the sandbox) and untrusted rendering engine. However, the presence of published client-side exploits for Chrome validates that such strategies are not a panacea. Recently, Barth et al. proposed a browser extension system that uses privilege separation and isolation to limit the impact of untrusted extensions [21]. The Polaris system uses the principle of least authority to restrict the impact of running untrusted applications [79]. BLADE's unconsented-content execution prevention is a similar concept to sandboxing. However, BLADE fully prevents the binary execution from occurring, rather than imposing privilege limitations, and it is significantly more transparent in how it uses user-dialog confirmation to auto-remap user-initiated downloads. More significantly, unlike secure browser frameworks that require the adoption of an entirely new browser, BLADE security protections can be deployed underneath the wide range of current and legacy Internet browsers.

2.8 Conclusion

We introduced the BLADE system as a new approach to immunizing vulnerable Windows hosts from surreptitious drive-by download infections. The BLADE system incorporates a kernel module to track all browser-to-human interactions, and then uses this information to distinguish consented web-based binary downloads from those cases where covert binary installations are performed. In the former case, the user-consented binaries are transparently remapped to the filesystem, and BLADE imposes no perceptible runtime behavioral changes or performance impacts on the browser. In the latter case, BLADE isolates and reports the malicious link and binary to the user, and unlike traditional sandboxes these malicious binaries are never executed.

We presented results from an ongoing evaluation of BLADE against thousands of active drive-by exploits currently plaguing the Internet (our evaluation results are unfiltered, auto-generated, and posted publicly to www.blade-defender.org). To date, BLADE's interception logic has demonstrated 100% effectiveness in preventing covert binary installations using the most widely deployed browsers on the Internet. Furthermore, over the past six months we have tested BLADE against the newest 0-day drive-by exploit attacks within days of their release and none have circumvented BLADE. In our next phase, we plan to extend BLADE support to other network-capable applications subject to drive-by download attacks.

CHAPTER III

SURF: DETECTING AND MEASURING SEARCH POISONING

Search engine optimization (SEO) techniques are often abused to promote websites among search results. This is a practice known as *blackhat SEO*. In this work we tackle a newly emerging and especially aggressive class of blackhat SEO, namely *search poisoning*. Unlike other blackhat SEO techniques, which typically attempt to promote a website's ranking only under a limited set of search keywords relevant to the website's content, search poisoning techniques disregard any term relevance constraint and are employed to poison popular search keywords with the sole purpose of diverting large numbers of users to short-lived traffic-hungry websites for malicious purposes.

To accurately detect search poisoning cases, we designed a novel detection system called SURF [58]. SURF runs as a browser component to extract a number of robust (i.e., difficult to evade) detection features from *search-then-visit* browsing sessions, as shown in Figure 5, and is able to accurately classify malicious search user redirections resulted from user clicking on poisoned search results.

Our evaluation on real-world search poisoning instances shows that SURF can achieve a detection rate of 99.1% at a false positive rate of 0.9%. Furthermore, we applied SURF to analyze a large dataset of search-related browsing sessions collected over a period of seven months starting in September 2010. Through this long-term measurement study we were able to reveal new trends and interesting patterns related to a great variety of poisoning cases, thus contributing to a better understanding of the prevalence and gravity of the search poisoning problem.

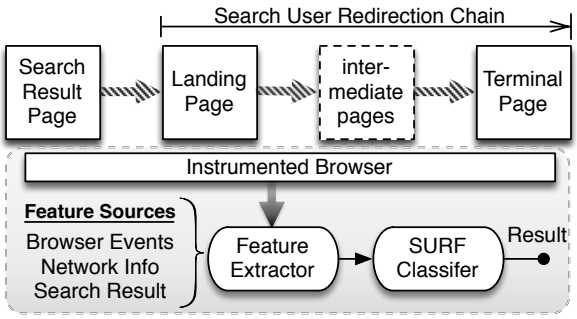


Figure 5: SURF Architecture

3.1 Background and Problem Study

In this Section, we provide a brief overview of the fundamentals of search engines and the reason why search results are subject to manipulation. We then present a study of search poisoning based on real world data, and discuss the observations that inspired our detection approach.

3.1.1 Search Engine and Blackhat SEO

eb search engines answer a user's query with a list of webpages selected from their index, in a descending order of relevance. After years of evolution, web search has grown into a relatively mature phase where major vendors are following similar work models.

Search engines typically employ crawlers to discover newly created or updated webpages. Each crawled page is then indexed based on keywords retrieved from its content. Upon a search query, webpages are ranked based on their relevance to the search terms and presented to the user.

This gives the abusers the following advantages. First, search engines implicitly trust the authenticity of the content on the indexed webpages, even though the content is under complete control of the website owners. Second, a web server can easily distinguish between search crawlers and human visitors. It is this *implicit trust* and *distinguishability* that give rise to blackhat SEO, whereby a web server fulfills webpage

requests from crawlers with specially crafted content having inflated relevance to a selected set of keywords, referred to as the *target keywords*. In addition, these crafted pages often contain large numbers of cross-reference hyperlinks to webpages that belong to the same blackhat SEO campaign. These hyperlinks have the effect of increasing the incoming link count for the promoted webpages, thus boosting the ranking in the search results.

Despite the fact that they involve some level of dishonesty, blackhat SEO techniques are sometimes used by legitimate businesses. In this paper, we distinguish between two types of blackhat SEO, namely *search inflating* and *search poisoning*. Search inflating aims to boost a website ranking through search keywords closely related to the promoted website, therefore only attracting users who search for topics related to the promoted website. On the other hand, search poisoning aims to boost a website's ranking though popular search keywords, regardless of whether these keywords are actually related to the promoted website's content or not. Unlike search inflating, which may be adopted by some legitimate websites, search poisoning only fits the need of visitor-hungry websites that simply want to increase the number of visitors for malicious purposes (e.g., for malware propagation purposes).

Most previous works on blackhat SEO detection apply lexical and structural analysis on page content [66, 82], or graph-based analysis on hyper-links [87]. However, very limited success has been achieved in practice [16], as the battle soon turned into an arms race. Having full control over webpage visibility and the freedom of adopting new evasion techniques gives adversaries significant upper hands, and makes it fundamentally difficult to design robust detection schemes based on lexical and structural analysis. To mitigate these problems, our approach aims to detect search poisoning instances using a set of features that are collectively difficult to evade because they are intrinsic to how search poisoning works, as we discuss in Section 3.2.3.

3.1.2 Search Poisoning Study

Search poisoning instances luring visitors to malware websites were first reported in 2007 [10]. However, until recently search poisoning has not been sufficiently studied, and has been only sporadically mentioned within the anti-malware community (mostly due to *fake AV* websites [72]).

To gain a more in-depth understanding of the search poisoning problem, we manually analyzed a dataset $\mathcal{S}_{study}$ containing 1,084 real-world search poisoning cases collected in September 2010. This preliminary study aimed to discover a set of robust features that can be leveraged for detection purposes, and to inspire our overall detection approach.

To collect the dataset $\mathcal{S}_{study}$ we proceeded as follows. We deployed an army of instrumented browsers, which on a daily basis automatically query Google and Bing with keywords that have been popular for the past 7 days. For each query, the browsers visited the top 100 URLs in the search results[1]. All network data and browsing events occurred during each browsing session were recorded as a browsing trace. This data collection process resulted in a very large dataset $\mathcal{D}$ containing over half a million browsing traces. $\mathcal{S}_{study}$ was derived from $\mathcal{D}$ using a simple heuristic to select traces that lead to malicious [6] or non-reputable webpages [8] with content irrelevant to the search keywords. This coarse-grained filtering yielded 1,084 highly likely search poisoning cases consisting of 596 unique landing URLs. It is worth noting that this dataset is not meant to be inclusive of all poisoning traces in $\mathcal{D}$, which is impossible to achieve without first developing a reliable detection system. However, our filtering heuristic produced a $\mathcal{S}_{study}$ dataset that exhibits satisfactory accuracy and sufficient diversity, as confirmed by our manual analysis. Therefore, $\mathcal{S}_{study}$ represents a reasonable base for our preliminary study of search poisoning.

[1] excluding URLs already flagged as malicious by search engines.

Below we itemize our observations and lessons learned from our manual analysis, which inspired the choice of the statistical features used by SURF.

O1: Ubiquitous use of cross-site redirections

We found that over 98% traces in $\mathcal{S}_{study}$ contain one or more redirections that cross website boundaries. The remaining 2% of browsing traces that do not contain such redirections are mostly due to incompletely rendered webpages, or modal dialogs that require non-trivial user interactions to proceed. On the other hand, less than 6% of the entire traces in $(\mathcal{D} - \mathcal{S}_{study})$ involve cross-site redirections. This ubiquitous use of cross-site redirections can be intuitively explained by the high risk and low effectiveness of exposing the malicious terminal website directly to search engines for rank promotion (thus a separated landing website is needed). For example, search engines have various security detectors in place to filter known malicious webpages and downgrade ranks of suspicious ones. Therefore, directly promoting malicious webpages can be a vain attempt and risks to jeopardize the entire search poisoning campaign. In fact, as our study went deeper, more evidence emerged supporting the need for malicious search user redirection in search poisoning.

O2: Search poisoning as a service

From all traces in S_{study} we extracted their chain of redirections, which are then used to compose a redirection graph, in which the nodes represent encountered domains and the directed edges represent redirections from one domain to another. Large numbers of inter-connected chains form subgraphs that represent different search poisoning campaigns. Two representative subgraphs are shown in Figure 6, as examples to illustrate our findings. Successful campaigns are able to employ many landing domains and target different search keywords to maximize the incoming search users. Figure 6 (top) shows a campaign that successfully poisoned over 28 "trendy" search keywords and injected at least 46 URLs into top search results. Furthermore,

the variety of terminal domains supported by a single campaign suggests that specialized search poisoning services are available to all kinds of malicious websites for purchase. The graph also indicates a two-tier affiliate marketing model followed by this campaign. Some landing pages redirected search users to centralized "super affiliate" domains (circled in the graph), which then dynamically dispatch the lured users to different terminal domains. As a result, more intermediate webpages appeared in the redirection chains.

O3: Sophisticated poisoning and evasion tricks

Cloaking techniques [44] are commonly used in search poisoning (by 97% landing pages in S_{study}). Search crawlers are presented with specially crafted content with fake relevance. The malicious redirection process only starts when visited by search users that queried the target (poisoned) keywords, while blocking other visitors as an attempt to prevent security detectors reaching the malicious content or domains. By forging the browser's `User-Agent` strings, we managed to obtain the search crawler views of 26 landing pages in our S_{study} dataset that did not verify the crawler's source IP address. These views were carefully composed to mimic normal webpages (e.g., blogs or news sites), with highly relevant content (possibly scraped from elsewhere) organized in a smooth way. We noticed that this well-crafted content may easily fool human readers, and is therefore very likely to evade content-based blackhat SEO detectors. In addition, we discovered a handful of image-rich landing pages that likely targeted at multimedia searches.

Another way in which search poisoning try to evade detection is by hosting the landing pages on compromised websites. These websites typically have been indexed by search engines for quite some time and accumulated a non-trivial domain history or reputation. This can help search poisoning to bypass some security checks performed by search engines and to facilitate rank promotion. In our study dataset S_{study} we found that about 70% of the redirections start from domains with a fair reputation

score in [8] and only 2% originate from blacklisted domains in [6].

O4: Persistence under transient appearances

To achieve a persistent poisoning effect, search poisoners have to accommodate for the volatility of popular search keywords. The bottom graph in Figure 6 shows a campaign that made multiple appearances on different popular search results across the entire study period: old landing domains had been active only for a limited time (a few days) before a new batch came in with a new set of poisoned search keywords. Rapidly rotating landing domains not only enable a wide coverage on trendy search topics, but also hinder detection efforts due to their transient appearance. Terminal domains behave in a similar way. An important difference is the fact that terminal domains tend to be disposable and have short registration periods (likely using domain tasting services), which further impedes blacklist-based detection.

O5: Various malicious applications

While previous reports always associate search poisoning with malware distribution websites[14, 50], search poisoning is used in a variety of other malicious applications. Figure 6 shows at least three types of malicious websites that use search poisoning to promote different types of cyber crimes, such as distributing fake AV software, hosting of rogue pharmacy sites, and other types of scams. Other types of uses (not reported in Figure 6) were also observed in S_{study}, such as sites that host drive-by download exploits, are part of click fraud schemes, or host phishing pages. Therefore, we argue that security solutions specific to individual types of malicious websites fail to thoroughly address the general search poisoning problem.

Lessons learned: Detecting search poisoning is a daunting task. Solely relying on identifying suspicious features associated with the landing pages (e.g., deceptive relevance, suspicious linkage structures, etc.) is immediately subject to evasion, given

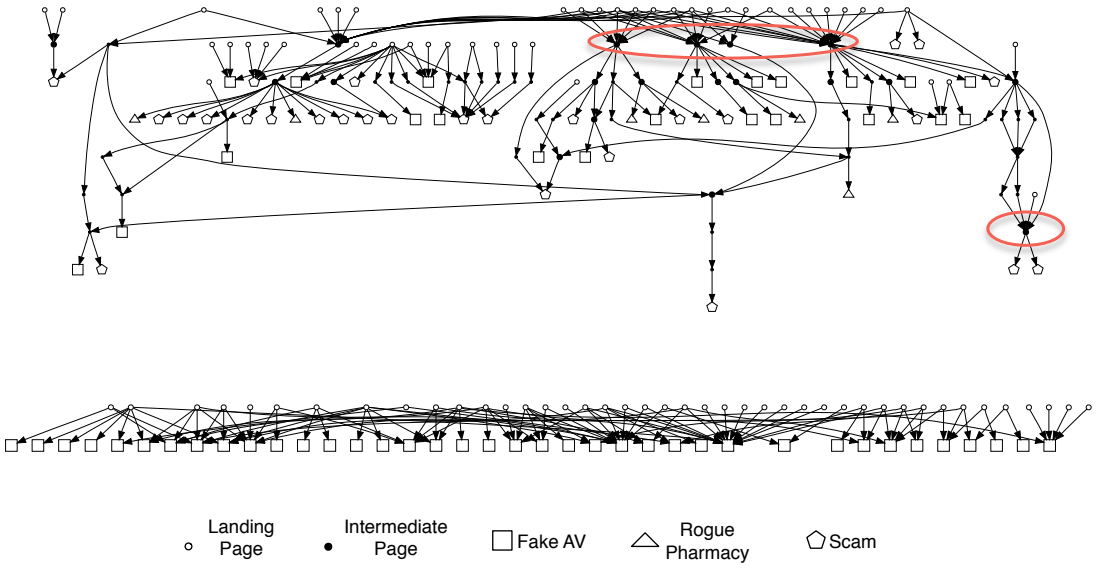


Figure 6: Redirection graphs of two search poisoning campaigns

the attackers' freedom to craft the page content. At the same time, detecting malicious terminal pages is hindered by their diversity and conditional accessibility of the actual malicious content (in particular, malicious sites can detect crawlers and security scanners). However, our observations convey a positive message to the defenders: the malicious search user redirections are intrinsic to search poisoning cases and exhibit distinguishable behaviors that are difficult to avoid completely for search poisoning to be successful. SURF's design was inspired by these findings.

3.2 SURF System Design

Based on the lessons learned, we set three primary design goals for our detection system:

- *Generality*: Search poisoning techniques are employed by attackers to promote a variety of malicious contents, and are not limited to luring users to visiting malware distribution pages. Therefore, SURF aims to detect generic search poisoning instances, regardless of the malicious content the attackers intend to promote.

- *Robustness*: While it is arguably impossible to completely prevent an arms race between defenders and attackers, we aim to identify features typical of search poisoning cases that are difficult to evade. In practice, we restrict SURF to using a set of *robust* features, which cannot be evaded by adversaries without incurring a significant cost (e.g., because of the need to completely change their attack strategy, or move to a different attack infrastructure).

- *Wide deployability*: Unlike most previous work on blackhat SEO detection, which is constrained by the dependency on search engine private data and only deployable at the search engine side, our approach aims to provide a solution that can be deployed at end-user's browsers (as a plugin), at automated security crawlers, and inside search engines. End-users can be protected from malicious terminal webpages hidden behind poisoned search results. At the same time, search engines or security vendors can deploy SURF in a "browsers farm" to accurately detect whether a search keyword is poisoned and harvest the malicious terminal page.

3.2.1 System Overview

To meet the goals listed above, we designed SURF as a browser component. An overview of SURF is shown in Figure 5. In practice, SURF sits in a browser and observes search-related browsing sessions. Whenever a user submits a query to a search engine and receives the result page, SURF starts its monitoring on page loads and redirections, from the search result page, to the landing page (on user's click), and to the terminal page the user is eventually brought to (after going through several intermediate pages in some cases). During this course, SURF extracts a number of statistical features from a range of sources, such as browser events, network information regarding the domain names and IP addresses involved in the redirection chain, and the search results themselves (see Section 3.2.2 for details). The resulting feature

Aspects	Features	Source*	Evasion Possibility
Redirection Composition	Total redirection hops	B	Low
	Cross-site redirection hops	B,N	Low
	Redirection consistency	B	Low
Chained Webpages	Landing to terminal distance	B,N	Low
	Page load/render errors	B,N	Low
	IP-to-name ratio	B	Medium
Poisoning Resistance	Keyword poison resistance	S	Low
	Search rank	S	Low
	Good rank confidence	S	Low

* B=browser events/data; N=network info; S=search result

Table 3: Feature selection

vector is then sent to the *SURF Classifier*, which is trained to distinguish between normal redirections and *malicious search user redirections*. In practice, the *SURF Classifier* is a supervised statistical classifier trained using a labeled dataset containing examples of redirection chains resulted from clicking on either legitimate or a variety of poisoned search results. It is worth noting that our definition of "malicious search user redirections" in this paper is restricted to redirections following poisoned search results. Detecting other types of malicious redirection is out of the scope of this work. While the vast majority of redirections used by the search poisoning cases we encountered during our study (Section 3.1.2) only change the URLs of webpages' top frames, SURF also covers the malicious redirections that occur within dynamic subframes (e.g., an `iframe`).

3.2.2 Detection Features

Inspired by our study and guided by our design goals, we identified a set of nine statistical features that capture the characteristics of generic search poisoning instances, and that are difficult for the attacker to evade without incurring a significant cost (e.g., the attacker would need to move to a new search poisoning strategy and infrastructure). The features extracted by SURF are divided in three groups as summarized in Table 3 and detailed below.

Redirection composition: This group of three features aims at capturing discrepancies between the legitimate and malicious search user redirections. The `total redirection hops` records the number of redirections that transport the visitor from the landing page to a terminal page, whereas the `cross-site redirection hops` counts how many of these redirections cross website boundaries. In SURF, a cross-site redirection is a redirection that brings from a domain d_1 to a domain d_2, where the *second-level* domains[2] of d_1 and d_2 differ (e.g., `www.cnn.com` and `blogs.cnn.com` share the same second-level domain, while `www.cnn.com` and `www.bbc.com` do not). As noted in Section 3.1.2, the vast majority of poisoned search results rely on cross-site redirections to transport search users to the malicious terminal pages hosted on covert domains. On the other hand, legitimate search user redirections rarely send incoming visitors away to another websites, simply because of the common incentive of keeping as many visitors as possible within their own domain. The `redirection consistency` feature captures whether a redirection is only visible to targeted search users. In legitimate search user redirections, users who arrive to the landing page through a search engine or though a direct link (e.g., by typing the same URL on the browser's address bar, or clicking the hyperlink on a "non-search" website) will be redirected to the same terminal page displaying relevant content. This is in contrast with search poisoning cases, in which typically only users that reach the landing page through a search will be redirected to the malicious content, while other visitors will be presented with non-malicious content in an attempt to evade some detection systems or manual analysis (see Section 3.1.2). To measure redirection consistency, SURF can first command the browser to directly visit the landing page (thus effectively stripping the `Referrer` field in the HTTP request, for example), and then allows the user's click on the search result link to go through so that the two obtained

[2]The second-level domain of a domain name `d.c.b.a` is typically defined as `b.a`, where `a` is called the *top-level* domain. We leverage Mozilla's public domain suffix list to take effective top-level domains such as `co.uk` into account.

sets of redirection events can be compared for consistency. When used at the search engine-level (e.g., in a "browsers farm"), SURF could perform this comparison on redirections obtained by visiting the landing page from different IP addresses to bypass some of the cloaking techniques discussed in Section 3.1.2 whereby the malicious content is not provided to IP addresses visiting the same landing page twice in a row. It is worth noting that such cloaking technique would not affect SURF's protection when deployed at end user's browsers. If the attacker refuses to offer the malicious content at the second visit (i.e., when SURF allows the user's click on a search result to go through), the user will not be exposed to the malicious content in the first place.

Chained webpages: This group of features measures three properties of the webpages involved in search user redirections. The `landing to terminal distance` feature measures the (approximate) geographical and topological distances between the landing page and the terminal page. In practice, to measure this distance we leverage information about the geo-location of the IPs where the two pages reside, the autonomous systems (AS) the IPs belong to, and the websites' domain names. The intuition is that malicious search user redirections always "travel" a long distance. The reason is that in search poisoning cases the landing page is usually hosted on a (likely compromised) website that belongs to a separate (usually legitimate) organization, while the terminal page is often hosted on a "bullet-proof" server provided by a different (usually not legitimate or boarder-line) organization often located in a different country.

The `page load/render errors` flags pages in the redirection chain that failed to load or render properly, due to exceptions or network errors. The intuition is that compromised pages are sometimes blacklisted or remediated, and the redirection chain to the malicious terminal page may end prematurely. The `IP-to-name ratio` feature represents the number of the redirection URLs that use an IP address (e.g., `http://192.168.0.1/index.php`) divided by the number of redirection URLs that

instead use a domain name (e.g., `http://example.com/index.php`). This is motivated by the fact that a large number of search poisoning cases encountered in our study involve URLs that use IPs that are dynamically assigned to unnamed hosts, in an effort to bypass URL blacklists commonly available in major browsers.

Poisoning resistance: This group of features measure properties of the search keywords and their corresponding search results. The `keyword poison resistance` quantifies the difficulty of poisoning search results under a given keyword. We measure this feature using publicly available information. The basic idea is straightforward: given a certain search keyword, the competitiveness of promoting a link higher in the result rankings is reflected by how prominent the top ranked webpages are under that keyword. We use Google's PageRank [68] to measure the prominence of a website. The `keyword poison resistance` of a keyword k is defined as the average PageRank value of the top 10 ranked websites obtained from the search results under k. In practice, the higher the value of this feature, the more prominent websites competing for the top rank positions, and thus the more difficult for an attacker to poison the keyword and force a link to a rogue landing page to appear higher in the ranking.

From our study dataset $\mathcal{S}_{study}$ (see Section 3.1.2), we noticed that the distribution of poisoned keywords across different search poisoning cases is skewed towards keywords with low `keyword poison resistance`. This result was somewhat expected, because keywords that are popular and yet easier to poison than others tend to attract the attackers' attention. Another feature we consider is the `search rank` of a landing page. The higher the rank of a landing page, the lower the probability that the result has been poisoned. This follows directly from our previous argument that top ranked pages are often prominent websites, making it difficult for search poisoners to promote their sites ahead of these prominent sites. The `rank confidence` feature

is computed by dividing the `keyword poison resistance` by the rank of a particular search result. The higher the `rank confidence`, the less likely that the result is poisoned.

3.2.3 Qualitative Robustness Analysis

Here we present a qualitative analysis of the robustness of SURF's statistical features against evasion attempts. A quantitative robustness analysis is discussed in Section 3.3.

Redirection composition: This group of features tries to capture the "search poisoning as a service" phenomenon discussed in Section 3.1.2. In practice, attackers often compromise several legitimate websites to host rogue landing pages, which use deceptive content to promote their ranks and at the same time redirect lured search users to the malicious terminal webpages. While in principle an attacker can attempt to evade this group of features, this would force the attacker to move to a completely different malicious network infrastructure in which all malicious redirections and terminal malicious pages are hosted on the same second-level domain, for example. In addition to incurring the significant cost of changing the malicious network infrastructure, this could also increase the risk of being detected by the compromised website's administrators, thus exposing search poisoning instances to a more prompt remediation, and end up sacrificing the established rogue landing pages which typically take considerable amount of time and efforts to mature.

Chained webpages: This group of features measure properties of the webpages involved in search user redirections. The `IP-to-name ratio`, while useful for classification in most practical cases, is not very difficult to evade because it only requires the attacker to register more domain names. However, we would like to emphasize that carrying an evasion attack on this feature will not significantly impact SURF's accuracy, as shown in the quantitative analysis in Section 3.3. The remaining two features are harder

to evade. The `landing to terminal distance` feature depends on the attacker's network infrastructure, and therefore the same arguments we made for the *redirection composition* features hold in this case. The `page load/render errors` may depend, for example, on pages (and domains) along the redirection chain that have been blacklisted or remediated. Therefore, this feature is not under the attacker's direct control and is difficult to evade.

Poisoning resistance: Since these three features are derived from the search result pages themselves with the help of public PageRank data, which is determined by the search engine algorithms and out of attackers' control. Therefore these features are difficult to evade.

3.2.4 Prototype Implementation

SURF only requires a limited amount of data to be collected during *search-then-visit* browsing sessions, and can be easily incorporated into a browser as a plugin. To demonstrate the effectiveness of our detection approach, we implemented SURF on top of an instrumented version of Internet Explorer 8. This instrumented browser leverages `mshtml.dll` for HTML parsing and rendering, and is able to listen for event notifications (e.g., used to identify subframe redirections) and peek into browsing data (e.g., HTML code and user visible content) at different rendering stages. In addition, SURF is capable of emulating simple user interactions during visiting sessions that require certain user input to proceed (e.g., clicking on a message dialogue box, activating a mouse-over action, etc.).

To perform our evaluation of SURF, we used several instrumented browser instances and instructed them to retrieve lists of popular (or "trendy") search keywords, query each keyword on both Google and Bing, and visit the top 100 search results for each query. Our evaluation data collection started in September 2010. We deployed our browser on 20 virtual machines which would run daily to query search

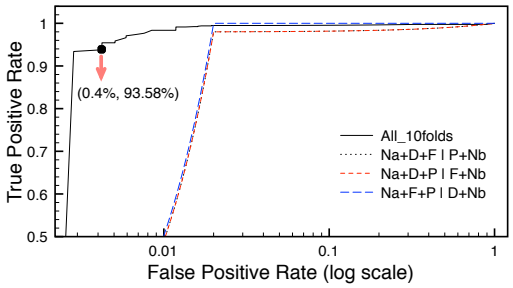


Figure 7: Threshold Curves (ROC)

keywords that had reached popularity within the past seven days. In addition, our instrumented browser was enhanced with BLADE [59], which we used to protect the VMs from *drive-by download* malware infections and to log attempted exploits. The obtained browsing information for each browsing session was then saved for offline analysis. While we performed our evaluation offline for convenience reasons, mainly to be able to run cross-validation experiments on large datasets, once the statistical classifier has been trained SURF can detect search poisoning instance *online* (see Figure 5).

3.3 Evaluation Results

3.3.1 Evaluation Dataset

To the best of our knowledge, there exists no public labeled dataset of search results and their related user redirection events related to search poisoning cases. Therefore, we chose to semi-manually label part of our dataset. In particular, we labeled browsing sessions collected during October 2010. In the following, we refer to obtain dataset as $\mathcal{S}_{eval}$. It is worth noting that $\mathcal{S}_{eval}$ differs from the $\mathcal{S}_{study}$ dataset that we used for the search poisoning study in Section 3.1.2 ($\mathcal{S}_{study}$ was collected one month earlier). This is to make sure that SURF does not "overfit" $\mathcal{S}_{study}$ because $\mathcal{S}_{study}$ inspired the choice of SURF's statistical features, and to avoid over-estimating SURF's accuracy. For the same reasons, none of the statistical features measured by SURF were used to guide our labeling. We simply semi-manually labeled the data using a separate set

of heuristics based on the relation between the search terms and a (visual) analysis of the terminal page content rendered by the browser.

In practice, we labeled search poisoning cases related to three types of malicious activities: search result for popular search keywords that lead to irrelevant terminal pages serving (1) *drive-by download* malware, (2) *fake AV* software, or (3) hosting *rogue pharmacy* sites. We labeled all these cases as *poisoned* (or positive). At the same time, we labeled a search result as *non-poisoned* (or negative) only when all URLs appeared in redirection chain have a fair reputations (e.g., according to [8]) and none of them are flagged as malicious by website scanning or blacklisting services (e.g, using [6]). Overall, $\mathcal{S}_{eval}$ consists of 1,184 negative samples and 1,160 positive ones, with 585 fake AV, 414 drive-by download, and 161 rogue pharmacy cases.

To evaluate SURF and confirm it follows our design goals, we conducted three different experiments. The first experiment aims to estimate SURF's accuracy, while the second attempts to show that SURF is able to detect generic search poisoning cases, and is not limited to one specific type of malicious content. The third experiment aims to show what features are the most important for classification, and how SURF may respond to evasion attempts on these features. Throughout all our experiments, we used Weka's `J48` decision tree classifier [45], which is an implementation of the well known C4.5 algorithm [9]. This choice is motivated by the fact that decision trees are efficient (both during the training and testing phases) compared to other statistical classifiers, and the resulting trained classifier can be easily interpreted.

3.3.2 Overall Accuracy

To estimate SURF's detection rate and false positives, we performed a 10-fold cross validated of SURF's classifier on the $\mathcal{S}_{eval}$ dataset, achieving an average true positive rate of 99.1% at a 0.9% false positive rate. The ROC curve in Figure 7 shows a very slow decrease of the detection rate as the false positive rate is pushed down from 0.9%

to 0.28%. Therefore SURF can satisfy a relatively wide range of usage scenarios with different levels of tolerance to false positives, while still maintaining a reasonably high true positive rate. In our SURF prototype we set the detection threshold to limit the false positive rate to 0.4% (marked point in Figure 7). We also analyzed the decision tree produced by the trained `J48` classifier and found that misclassifications were mainly caused by those rare cases in which poisoned search results achieved a top rank under a very competitive keyword, or cases in which a legitimate landing page redirects visitors to a very "distant" terminal page (see Section 3.2.2) and detours sampled visitors through third-party traffic analysis services.

3.3.3 Generality Test

To confirm that SURF is able to detect generic search poisoning cases, regardless of the specific malicious content that they promote, we performed the following experiment. We prepared three datasets, $\mathcal{D}$, $\mathcal{F}$, and $\mathcal{P}$, containing labeled search poisoning example from the drive-by malware downloads, fake AV, and rogue pharmacy cases from $\mathcal{S}_{eval}$, respectively. In addition, we prepared two separate datasets, $\mathcal{N}_a$ and $\mathcal{N}_b$, containing randomly selected negative examples (i.e., legitimate search redirection cases). We then performed a 3-fold cross validation by using $\mathcal{D} \cup \mathcal{F} \cup \mathcal{N}_a$ for training, and testing on $\mathcal{P} \cup \mathcal{N}_b$ for the first fold, training on $\mathcal{D} \cup \mathcal{P} \cup \mathcal{N}_a$ and testing on $\mathcal{F} \cup \mathcal{N}_b$ for the second fold, and training on $\mathcal{F} \cup \mathcal{P} \cup \mathcal{N}_a$ and testing on $\mathcal{D} \cup \mathcal{N}_b$ for the third one. Figure 7 shows the three separate ROC curves (one per fold). Averaging the results of the 3-fold validation, we obtained a detection rate of 98% at a false positive rate of 1.8%. It is worth noting that while this result does not appear to be as good as the result obtained for the overall 10-fold cross validation experiment, our 3-fold evaluation presents SURF with much harder cases in which no examples of search poisoning instances of the same category used for testing are present in the training set. However, the high detection rate and relatively limited false positives

demonstrate that the features used by SURF can indeed capture generic search poisoning properties, independent of the specific type of malicious content delivered by the terminal page.

3.3.4 Feature Robustness

This experiment attempts to quantify SURF's resistance to evasion effects on the statistical features. To perform the experiments, we ran SURF's classifier on 100 randomly chosen positive samples. We artificially modified the values of the features describing these 100 samples to simulate different evasion scenarios, and evaluated how the classifier's accuracy changed as a consequence. We first artificially set the `IP-to-name ratio` to zero, which is the most common value of the feature in negative samples. The `IP-to-name ratio` feature is the only one among SURF's features that is not difficult to evade. After altering this feature, only 1 out of 100 samples was misclassified, which suggests that the attacker cannot gain much by attempting to evade it. Despite the fact that other features are hard to evade, we nonetheless wanted to investigate the effect of altering the most discriminant features, i.e., the features that appeared close to the root of the decision tree. The `redirection consistency` and `landing to terminal distance` turned out to be the top two most discriminant features. Replacing their values with values drawn from negative samples caused 80 out of 100 samples to be misclassified. However, it is worth noting that evading these two features would require the attacker to change her malicious network infrastructure, thus incurring a significant cost, as discussed in Section 3.2.3

3.4 Discussion

At a first glance, our evaluation may seem unconvincing because we used a dataset labeled by ourselves. However, to the best of our knowledge, no public labeled dataset exists that contains browsing session data related to a variety of search poisoning instances. As discussed in Section 3.3, our semi-manual labeling process is based on

a set of heuristics that do not overlap with any of the detection features measured by SURF. This allowed us to perform an unbiased evaluation. While we acknowledge that our semi-manual labeling can only help us collect partial "ground truth", it is extremely difficult, if not impossible, to obtain complete ground truth without a deterministic search poisoning detection system. In absence of such perfect search poisoning detector, our labeling method represents our best effort to produce a representative (even though not complete) ground truth that includes a variety of search poisoning instances leading to different types of malicious content.

During our feature selection process, we discarded a few candidate features that may help the classification accuracy but are not robust. For example, we chose not to include features based on measuring the relevance of the content of terminal pages to the search keywords because the content of the terminal pages is under complete control of the attacker, making these types of features easy to evade. Also, we did not include features related to the structure of the URLs involved in malicious search user redirections. These features usually require historical knowledge of the "normal" structure of the URLs for each particular website. While these type of features may be included in a search engine-side deployment of SURF, client-side deployments would not be able to collect and leverage this kind of information, and therefore we decided not to add them to our prototype implementation. Furthermore, an attacker has a non-negligible flexibility when choosing the structure of the redirection URLs, and therefore it is not clear how robust these features would be. We also considered some features based on domain or IP reputation scores. Though capable of reducing the false positive rate, these features were excluded from our selection because of their heavy dependence on external security services, and because we wanted to evaluate the detection accuracy of SURF based solely on the strengths of our own features. In practice, SURF implementations may opt to incorporate reputation-based features to improve classification accuracy.

When deployed at the search engine side (e.g., using a "browser farm"), SURF can be used to analyze suspicious search results and accurately detect poisoning cases. This can provide search engines with valuable information that goes beyond specific poisoning instances. For instance, the landing pages involved in search poisoning are often organized in a "botnet mode", so that the keywords to be poisoned can be periodically fetched from a command-and-control server. Therefore, detecting search poisoning cases can reveal information about compromised websites and botnet organizations. In addition, newly detected malicious terminal webpages may serve as labeled samples for malicious webpage detectors that require periodic re-training.

At the same time, SURF can be deployed at each single client to detect (and block) poisoned search results. A possible deployment scenario could include large numbers of client-side SURF installations that collaboratively detect search poisoning case and share information about the underlying malicious network infrastructures (e.g., domain names, IP address, etc.) through a cloud service, thus potentially improving SURF's detection accuracy.

3.5 Empirical Measurements

To gain a deeper insight into the search poisoning problem, we performed a long-term (7 months, 212 days) measurement study. We manually analyzed data in the first month. We used SURF to analyze browsing data we collected in the next 6 months. During this period we instructed SURF to analyze over 12 millions search results from both Google and Bing collected by querying the top 40 "trendy" keywords [4]. In practice, once a search keyword appeared in the top 40 list on a given day, we used SURF to query the search engines for this keyword for the following 7 days. Overall, during our measurement study SURF automatically queried the search engines with 8,480 keywords. This long-term data collection process enabled us to study in-the-wild search poisoning instances from two different angles, a "micro measurement"

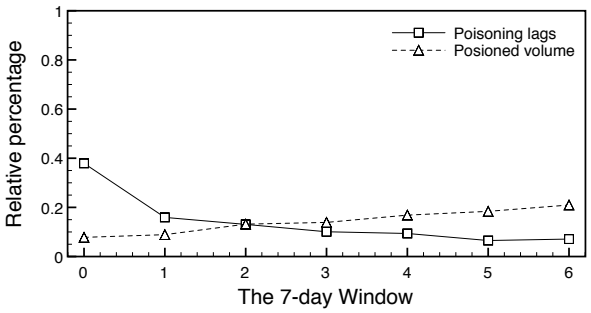


Figure 8: Micro measurement statistics

study based on a 7-day window that focuses on how search poisoning evolves with respect to frequently changing keyword popularity, and a "macro measurement" study that looks at poisoning trends over the entire 7-month period.

3.5.1 Micro Measurements

Due to frequent changes in the trendy search keywords [4], we expected that some time (e.g., a few days) would be required for the attackers to poison the search terms and make their landing pages of choice appear in the related top search results. Surprisingly, our measurements on the micro developments of detected search poisoning cases suggest otherwise: adversaries are extremely responsive and have built effective approaches to promote rogue landing pages under the targeted trendy search keywords in a short time.

Among the 3,869 keywords for which we detected related poisoned search results, 38% of them had *poisoning lag* (i.e., the time it takes for the first poisoned result to be detected) of one day or less. This percentage decreases as the lag increases, and only 7% of the keywords had a poisoning lag of 7 days, as shown in Figure 8. This results suggest that the adversaries are capable of keeping up with search users' interests, and that the majority of their poisoning attempts succeed within the first 3 days.

We also found that the average life time of a rogue landing page involved in search

poisoning is only 1.7 days. This indicates that adversaries favor a fast-switching strategy to reduce the exposure window, thus reducing the possibility of the rogue pages being detected and conserving the compromised landing sites for reuse in future poisoning attacks. However, the appearance of these rogue landing pages in the search results lasts for more than 3 days on average, until the page ranking is demoted due to the new information retrieved by the search crawlers. At the same time, the relative volume of detected rogue landing pages for a given poisoned keyword keeps increasing throughout the 7-day observation window, as shown by the *poisoned volume* increase in Figure 8 (the poisoned volume is relative to the total number of detected poisoned results during the 7-day period). We believe this fast paced operation proposes significant challenges for blacklisting and other traditional security solutions, making them inadequate to solve the search poisoning problem. In fact, well-known malicious webpage scanners (e.g., [6]) have failed to detect 78.9% of malicious terminal pages involved in the search poisoning cases detected by SURF (we scanned all the terminal pages using [6] on the same day when SURF detected the related search poisoning instance).

Our measurements show that the visiting traffic reaching a particular malicious terminal page is always contributed by multiple landing pages that appear in poisoned search results related to different search keywords. In particular, we found that during a 7-day period, for each given malicious terminal page, their visitors were supplied in average by 2.9 poisoned search keywords and 2.2 different rogue landing sites per keyword. We speculate this is a reflection of a growth in the "search poisoning as a service" phenomenon discussed in Section 3.1.2.

3.5.2 Macro Measurements

After examining the development of poisoned search results in the first 7 days period, we now zoom out our measurement window to consider the entire 7-month data

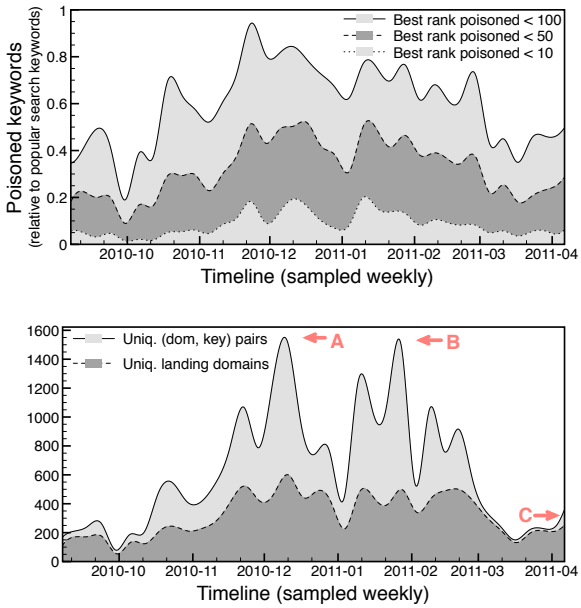


Figure 9: Poisoned keywords percentage (left) and landing domains engaging search poisoning (right)

collection period. Through these long-term measurements we aim to discover search poisoning's evolving trends and characteristics that are observable only throughout a long period of time. To highlight specific patterns and long-term trends, we divide the 7-month observation period in 31 epochs, where each epoch is equal to one week. Then, for each week we compute a number of statistics (discussed below), and plot how these statistics vary with respect to time.

During most epochs, we found that more than 50% of the search keywords that became popular on that epoch got poisoned. For this particular measurement, a search keyword is considered to be poisoned if at least one out of the top 100 search results for that keyword is related to a search poisoning instance. Figure 9 (left) plots the percentage of poisoned keywords for each week, broken down into different degrees of success in terms of search ranking. We can see that during some of the epochs, almost 60% of the trendy search keywords resulted in rogue landing pages that ranked within the top 50 search results. Furthermore, in some cases adversaries managed to promote their rogue landing pages up to the top 10 search results. These particularly successful attacks were related to about 15% of all poisoned keywords on

average. These findings are alarming because they suggest that a large number of search users can easily be affected by search poisoning.

Figure 9 (right) shows two curves. One represents the number of rogue landing sites (counted as the number of distinct related domain names) that were involved in search poisoning cases, and the other the number of distinct (*landing domain*, *keyword*) pairs. The two peeks (marked by A and B in the figure) that appeared around Christmas time and the Super Bowl are not a coincidence. Our analysis shows that important predictable events can help adversaries to further increase their poisoning success rate, given the sufficiently large preparing time (enabled by the predictability of the events) and the interest of large number of search users in the events being exploited. At the same time, less predictable breaking news that receive broad attention for a not too short amount of time (e.g., a few days) are also an easy target for search poisoning. An example of this is the earthquake and tsunami that recently hit Japan (marked by C in Figure 9). Furthermore, as the targeted keywords (upper curve) fluctuated between attempts to leverage different events, the number of detected landing domains (lower curve) remained somewhat more stable, suggesting that search poisoning operators have a solid footing in the search engines' indexes, and are ready to launch new attacks whenever the opportunity comes.

On average, users who fall victim of search poisoning are redirected at least twice, including one cross-domain redirection, before reaching the malicious terminal page. About 29% of these redirections were due to `HTTP` 30x responses. Not surprisingly, the majority of the remaining 70% were mostly due to client-side scripts (likely an attempt to evade security crawlers that do not support script execution). About 78% of the landing page URLs explicitly contained the targeted search keywords to boost the page relevance perceived by search crawlers. About 98% of the intermediate URLs include ID-like parameters to track unique visitors, identify search poisoning affiliates, or prevent repeated visits. In 94% of the cases, the terminal page URLs

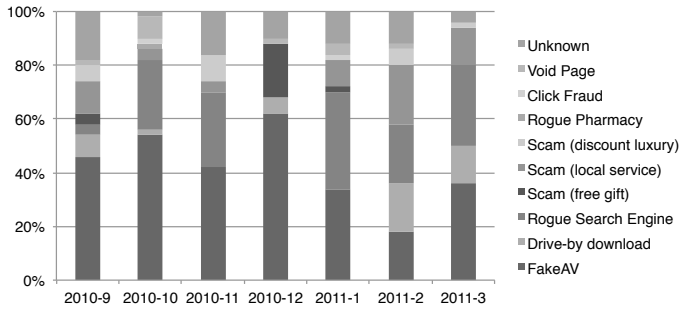


Figure 10: Terminal page variety survey

used domain names registered for less than a year, with many of these domain chosen to purposely deceive victim users and promote specific scams. Among the detected search poisoning cases, the most frequently used top-level domains (TLDs) for landing pages are `.com`, `.org`, `.net`, and `.info`, with a TLD distribution similar to regular websites. On the other hand, domains related to terminal pages were mostly registered under TLDs such as `.cc`, `.com`, `.in`, and `.net`, some of which are known to be malware friendly.

To have a sense of the variety of malicious content promoted by search poisoning, we surveyed 350 randomly chosen terminal pages. These 350 terminal pages were evenly distributed across our 7-months measurement period. The results are shown in Figure 10. We manually categorized the terminal pages based on data saved at the time when the page was visited by SURF, including screen shots of each rendered page. As expected, fake AVs are the most prevalent adopters of search poisoning during the entire 7-month period. However, we noticed that their pervasiveness started fading as other types of malicious terminal pages increased. Drive-by malware downloads and other browser exploitation techniques did not appear to be commonly leveraging search poisoning. On the other hand, various types of social engineering-based malicious pages are dominant players. The figure shows a clear surge of rogue search engine pages, which present the users with links seemingly relevant to the search keyword but aim to profit from user clicks. Scam pages (e.g., watch replicas, etc.) represent another significant fraction of the surveyed terminal pages. Regardless of

their individual tactics, scam pages in general bait traps with free or unrealistically cheap goods to attract users and steal private information (e.g., credit card numbers, passwords, etc.). We also encountered a number of malicious terminal pages related to click fraud and rogue pharmacies. The terminal pages categorized as "void" typically contain clues of of certain types of maliciousness (e.g., based on their domain name patterns) but were inaccessible when visited due to unsuccessful DNS resolution, or webpage errors. SURF's ability to detect even these "void" malicious terminal pages supports our initial goal of building a detection system that is agnostic to the specific content of malicious pages promoted through search poisoning.

3.6 Related Work

Blackhat SEO countermeasures: Blackhat SEO, which involves abusing search engine optimization techniques to achieve undeserved rankings, is not a new problem and has been studied for year, especially in the information retrieval community. Most proposed detection methods work at the search engine level and attempt to identify deceptive information introduced by the adversaries into the search index to influence the rankings of their websites. Various detection features explored by these methods mainly focus on two aspects of indexed webpages: intra-page characteristics [66, 82, 88] and inter-page linkages [87]. However, for adversaries with full control over their injected search landing pages, such features are not difficult to evade, sometimes even without requiring changes to their operation routines. In fact, this traditional way of countering blackhat SEO has failed to stop its rising trend [17]. SURF addresses search poisoning, a new class of blackhat SEO, building on the lessons learned from previous work and approaching the detection from a new angle using a set of feature that is more robust to evasion.

deSEO[52] is a very recent work done in parallel with SURF. It detects URLs from the search index that contain signatures derived from known search poisoning

landing pages and exhibit patterns not previously seen by the search engine on the same domain. Since there is no need to crawl each URL, this approach scales much better than SURF when facing a huge volume of search results. However, deSEO is limited by the coverage of the URL signatures, and may only find a subset of what SURF detects. For example, about 12% landing page URLs detected by SURF in Section 3.5.2 do not contain trendy search keywords, and thus may be missed by deSEO. Moreover, SURF does not rely on any information internal to search engines and can be deployed at the client side, enabling single browsers to detect poisoned search results as well as malicious webpages behind them before the content is presented to the user.

Malicious webpage detection: SURF, when implemented as an automated detection agent, can be viewed as a dynamic crawler used to scan search results looking for poisoned ones. From this perspective, SURF is similar to many proposed systems that crawl the Internet for various kinds of malicious webpages [63, 84]. Such systems always employ an army of browsing agents running in a controlled environment to visit suspicious URLs in batch and detect signs of specific types of malicious activities. SURF can be easily integrated into these systems and can enable the detection of search poisoning cases along with the related compromised landing page and malicious terminal pages. On the other hand, solely relying on malicious page detectors for finding poisoned search results may achieve limited success, because of the variety of terminal pages, many of which use social engineering attacks that are difficult to detect.

Applying machine learning techniques to data collected during a crawling session is also a common approach to detecting malicious webpages. A recent work [80] is able to detect URLs that lead to spam pages. Our work is different because SURF is not limited to detecting spam pages, and can instead detect generic search poisoning cases.

3.7 Conclusion

Search poisoning is an abuse of SEO techniques by which miscreants target any search term that can maximize the number of incoming search users to their malicious websites. We observed through an empirical study that a key characteristic of search poisoning is the ubiquitous use of cross-site redirections. We designed and implemented SURF, a novel detection system that runs as a browser component and is able to detect malicious search user redirections resulted from user clicking on poisoned search results. SURF extracts a number of detection features from *search-then-visit* browsing sessions. These features are robust and the resulting classifier is hard to evade because they capture the key properties of search poisoning (derived from our empirical study and analysis). Our evaluation showed that SURF can achieve a detection rate of 99.1% at a false positive rate of 0.9% on a dataset that contains real-world search poisoning instances. Using SURF, we also performed a long-term measurement study on search poisoning on the Internet over a period of seven months. Our study revealed new trends and interesting patterns related to a great variety of poisoning cases, and underscored the prevalence and gravity of the search poisoning problem.

CHAPTER IV

CHEX: STATICALLY VETTING ANDROID APPS FOR COMPONENT HIJACKING VULNERABILITIES

An enormous number of apps have been developed for Android in recent years, making it one of the most popular mobile operating systems. However, the quality of the booming apps can be a concern [5]. Poorly engineered apps may contain security vulnerabilities that can severally undermine users' security and privacy. In this paper, we study a general category of vulnerabilities found in Android apps, namely the component hijacking vulnerabilities. Several types of previously reported app vulnerabilities, such as permission leakage, unauthorized data access, intent spoofing, and *etc.*, belong to this category.

We propose CHEX [57], a static analysis method to automatically vet Android apps for component hijacking vulnerabilities. Modeling these vulnerabilities from a data-flow analysis perspective, CHEX analyzes Android apps and detects possible *hijack-enabling flows* by conducting low-overhead reachability tests on customized system dependence graphs. To tackle analysis challenges imposed by Android's special programming paradigm, we employ a novel technique to discover component entry points in their completeness and introduce *app splitting* to model the asynchronous executions of multiple entry points in an app.

We prototyped CHEX based on Dalysis, a generic static analysis framework that we built to support many types of analysis on Android app bytecode. We evaluated CHEX with 5,486 real Android apps and found 254 potential component hijacking vulnerabilities. The median execution time of CHEX on an app is 37.02 seconds, which is fast enough to be used in very high volume app vetting and testing scenarios.

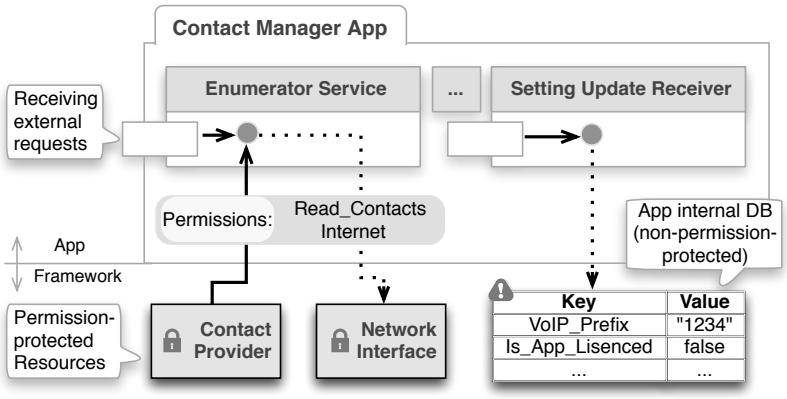


Figure 11: An app vulnerable to component hijacking

4.1 Component Hijacking Problem

Android framework dictates a component-based approach to app design, for flexible interoperability among apps and efficient app lifecycle management. In this approach, app developers organize their code into individual *application components* [19] (*i.e.* `Activities`, `Services`, and etc.). Each component fulfills a logically independent task and can serve requests from other components in the same app, the framework, or another app if the component is publicly available (or is *exported*, in Android terminology). For example, an instant messaging app may need a contact enumerator (*i.e.* collecting all contacts on the device) to suggest friends for the user. Instead of implementing its own, the app can leverage an existing contact enumerator component exported by a contact manager app.

However, the capability of reusing a component under its containing app's identity can lead to serious security threats, when the component is security-critical but not well protected. To generalize threats of this kind, we introduce the concept of *component hijacking*, describing a class of attacks that seek to gain unauthorized access to protected or private resources through exported components in vulnerable apps. As shown in Figure 11, if the contact manager app fails to deny requests from unauthorized apps, a malicious app can easily take advantage of (or hijack) the `Enumerator`

`Service` and consequently gain access to user's contacts without the required permission. Recent works [29, 55, 39, 41] reported attacks similar to this particular example, all of which derive from the classic confused deputy attack [46] and aim at escalating privileges of attacking apps in the context of Android's permission system. Note that, although permission-protected resources (*e.g.* contacts, geo-location, and *etc.*) are obvious targets, component hijacking is by no means limited to these confused deputy attacks that bypass the permission checks. In fact, if carelessly exposed, data or invokable interfaces that are only intended for app's internal use (thus not permission-protected) can also become targets of component hijacking attacks. In this case, where no explicit permission is involved, the attacking app seeks to tamper or steal private data of a vulnerable app that does not enforce access control or input validation properly. For instance, in Figure 11, the security-critical information stored in the app internal database can be tampered through the `Setting Update Receiver` in an SQL-injection fashion. Complicated cases exist, where an attacker can leverage a chain of vulnerable components to steal private data, modify critical settings, or perform privileged actions, by simply issuing crafted requests as a regular app.

Several topics related to component hijacking were studied by recent works. ComDroid [28] checks app metadata and API usages for publicly exported components. Such components, if granted direct or indirect access to sensitive resources, may become launching points for hijacks. Grace *et al.* [41] analyzed factory stock apps to identify permission leakage, a threat that also spurred studies on its runtime mitigations [39, 30, 24]. While these works are effective in archiving their own goals, they target at the vulnerabilities that only represent a subset of component hijacking (*i.e.* hijacks seeking to access non-permission-protected sensitive resources are not covered). Plus, these works do not intend to provide any in-depth detection method suited for scalable app vetting. Our work aims to bridge this gap.

It is noteworthy that component hijacking vulnerability is not caused by any insecurity intrinsic to Android framework. In fact, Android does provide a set of mechanisms to secure app components and their interactions. Instead, similar to other security vulnerabilities in software, component hijacking stems from issues that are hard to avoid in reality, such as undertrained developers, lacks of proper app quality assurance, and usability issues of existing security mechanisms. We expect component hijacking vulnerability to emerge rapidly in terms of popularity and severity. As the user population of Android constantly grows, more and more developers are migrating to this platform, often with inadequate experience or knowledge on its security mechanisms. In addition, the current app distribution model offers a convenient way for amateur developers to release their apps to a wide range of users. With these factors adding up, the odds becomes high for a regular Android user to install apps that insecurely handle external requests and thus are subject to component hijacking. Attackers who are now struggling with crafting new exploiting techniques on Android would not easily let this new attacking vector pass by.

Apps with component hijacking vulnerabilities are generally not malicious on their own, but can be coerced by attacking apps to conduct malicious activities. Defensive efforts may focus on either finding the vulnerabilities in benign apps, or detecting corresponding exploits from suspicious apps. Our work follows the first approach for the more distinguishable and less volatile nature of the subject being detected, than that of the second one. Without loss of generality, we define component hijacking attacks as follows:

Definition 1 *An unauthorized app, issuing requests to one or more public components in a vulnerable app, seeks to:*

G1 : READ *sensitive data out of the app; or*

G2 : WRITE *to critical data region inside the app; or*

G3 : perform a combination of G1 and G2.

Based on this definition, to determine if a given component (or set) is vulnerable to hijacking is equivalent to finding feasible data flows that can enable any of the three goals above without going through any security checkpoint. We refer to these flows as *hijack-enabling flows* hereafter. In a simple example, the `Emulator Service` in Figure 11 is vulnerable if a hijack-enabling flow exists that fulfills *G1*: the flow propagates the contact list into an object to be returned to the requestor, serving as a data sink from which the requestor (or attacking app) can read data directly. In a more complex scenario, the data sink may not seem immediately accessible to the requester (*e.g.* sending contact to an URL, as shown in Figure 11). However, if the component contains a hijack-enabling flow that writes requester-supplied input into certain output-controlling data (*e.g.* the destination URL), requestors can still indirectly read the contact information by redirecting the output and achieve *G3*. Component hijacking is also possible on a chain of components, when the hijack-enabling flows span across component boundaries.

Defining component hijacking from a data flow perspective allows us to transform the vulnerability detection problem into an equivalent data flow analysis problem. A different but related topic is data leakage detection [34, 48], which looks for individual data flows that indicate sensitive data being propagated out of certain containment scope. Note that apps sending out sensitive data are not necessarily exploitable nor harmful (*e.g.* an app sends users' GPS information to remote servers for location-based services). Therefore, data leakage detection only reports outbound sensitive data flows without clarifying their security implications. In contrast, component hijacking vulnerability is always exploitable and undermines user's privacy. On the other hand, techniques for identifying component hijacking vulnerability can be applied to finding data leaks, but not vice versa. Because finding data leaks are essentially identifying special hijack-enabling flow that enable *G1* with all data sinks supposed to be accessible by attackers.

Component hijacking gives attackers the freedom to surreptitiously perform privileged actions and access private data. In our *threat model*, successful hijacks require users to willingly install the attacking app on their devices. To create a user population of decent size, attackers can resort to many illicit techniques that promote their apps in the market and lure users. Given component hijacking apps often requesting little to no permissions, users, even vigilant ones, tend to trust them easily. Although attackers cannot control, but only hope for, the availability of vulnerable apps on users' devices, the reality has been working towards attacker's favor due to the large number of under-trained Android developers and an overall lack of app quality assurance. Therefore, as a defensive effort, we designed CHEX to assist apps developers, testers, and market operators in filtering out apps vulnerable to component hijacking attacks before they reach end user devices. We chose to target CHEX on non-malicious apps, which constitute the majority of exploitation targets, so that we can safely assume a non-adversarial application scenario (*e.g.* heavy obfuscations and anti-analysis techniques are out of our concern) and solely focus on designing the detection and analysis method.

4.2 Detection and Analysis Method

CHEX follows a static program analysis approach, featuring a novel data-flow analyzer specially designed to accommodate Android's special app programming paradigms. Static analysis makes sense for vetting benign apps in that, the anti-analysis techniques that are commonly used in adversarial scenarios are out of scope, and the advantages of static analysis, such as its completeness and bounded time complexity, are well suited to addressing the vulnerability discovery problem.

Existing data-flow analysis and modeling methods are not immediately applicable to Andriod apps due to Android's special event-driven programming paradigm. Our

flow- and context-sensitive analyzer, incorporated with a number of analysis techniques and models that we devised for Android apps, can efficiently discover data flows of interest within the entire app. Its underlying flow extraction mechanism is separated from the high level policies that define interesting flows. As a result, our data-flow analysis method can be applied to other applications than vulnerability discovery. Our method also offers the flexibility to choose if the Android framework code[1] needs to be included or simply modeled during the analysis, depending on specific usage scenarios. In this paper, we model the framework code for reasons discussed in Section 4.3.

Next, we present a concrete example to illustrate component hijacking vulnerabilities, as well as typical challenges associated with performing data-flow analysis on Android apps.

4.2.1 A component hijacking example

Our example is a hypothetical Android app that aggregates the popular location-based services and provides a one-stop solution for users. Figure 12 shows a critical `Service` component of the app. Upon requested by particular `Intents`, this component obtains user's location information and synchronizes it with a remote server. Despite that the component is intended for the app's internal use only, its developer carelessly left it open to other apps. This mistake is not uncommon partly because Android by default publicly exports components that register to accept particular `Intents`. Here, we demonstrate two possible component hijacking attacks on this example app and highlight the challenges associated with analyzing the code. The vulnerabilities in this example app are similar to those that we found in the real apps and reported in Section 4.5.3.

In Figure 12, Method `onBind` (Ln. 5) is invoked by the framework whenever a

[1] Android framework consists of the Dalvik runtime and Android system libraries. We refer to it as the framework hereafter. Note that apps (including system apps) are not part of the framework.

```
1  public class SyncLocSrv extends Service{
2    Location currLoc;
3    final Messenger mMessenger = new Messenger(new
         ReqHandler());
4
5    public IBinder onBind(Intent intent) {
6      return mMessenger.getBinder();
7    }
8
9    private class ReqHandler extends Handler {
10     public void handleMessage(Message msg) {
11       ...
12       switch (msg.what) {
13       case MSG_UPDATE_LOCATION:
14         // get GPS location
15         currLoc = lm.getLastKnownLocation(PROVIDER);
16         break;
17       case MSG_SYNC_LOCATION:
18         // sync GPS with specified URL
19         String url = msg.getData().getString("url");
20         String[] sendParams = new String[] {url};
21         new SendToNetwork().execute(sendParams);
22         break;
23       ...
24       default:
25         ...
26       }
27     }
28   }
29
30   private class SendToNetwork extends AsyncTask<String
         , String, String> {
31     // run in a separate thread
32     protected String doInBackground(String[] params) {
33       HttpClient hc = new DefaultHttpClient();
34       HttpPost pst = new HttpPost(params[0])); // URL
35       pst.setEntity(new StringEntity("gps:"+currLoc));
36       HttpResponse resp = hc.execute(pst);
37       return resp.toString();
38     }
39   }
40   ...
41 }
```

Figure 12: Vulnerable component example

requester component connects to the `Service`. Android programming paradigm dictates that apps organize their logic into components of different kinds, whose lifecycles are managed by the framework in an event-driven manner. Each component implicitly or explicitly registers event handlers (*e.g.* Ln. 5, 10, and 32). These handlers serve as the entry points through which the framework starts or activates the component when handled events happen. Apps, even average-sized ones, can have a large amount of entry points of diverse object types and appearances, which posed the first challenge to our analysis:

C1 : Reliably discovering all types of entry points (or event handlers) in their completeness.

Method `onBind` returns to the requester component an object that implements the `IBinder` interface (Ln. 6) — a common pattern to achieve inter-component communications in Android apps. The requester component can then send messages for the `Service` to handle via the object. It is the framework that delivers the message and invokes `handleMessage` as an entry point (Ln. 10) when an incoming message arrives. Since the invocations of different entry points in an app can be asynchronous, we faced the second challenge:

C2 : Soundly modeling the asynchronous invocations of entry points for analysis.

Once connected to the example `Service`, an attacking app can exploit at least two separate component hijacking vulnerabilities to obtain the device location and perform network communications respectively, neither incurring any permission violations or user interactions. Specifically, the attacking app can send a `MSG_UPDATE_LOCATION` message, followed by a `MSG_SYNC_`
`LOCATION` message, to coerce the message handler to first retrieve the device location (Ln. 15) and then send the data to a URL of the attacker's choice (Ln. 21). Alternatively, using a single `MSG_SYNC_LOCATION` message, the attacking app is able to make connections to arbitrary URL he supplies in the message. Based on Definition 1, these two particular cases of component hijacking are enabled by data-flows that respectively allow the attacker to (i) read the location data (*i.e.* realizing *G1*), and (ii) write to the variable that controls the URL to be contacted (*i.e.* realizing *G2*).

Sometimes it takes multiple individual data-flows, loosely connected or partially overlapped, to enable one of the three goals described in Definition 1. In our example, two individual data-flows together allow the attacking app to read the location information (*i.e.* by forcing the vulnerable component to retrieve and send the location information to a specified URL): one flow carrying location data obtained on Ln. 15

to the `HTTP Post` on Ln. 36 and the other carrying requester-supplied URL on Ln. 19 to the same `HTTP Post` operation. To detect such hijack-enabling flows, a data-flow analyzer needs to tackle the challenge of:

C3 : Assessing the collective side-effects of individual data-flows and identifying converged flows of interest.

For optimized responsiveness, Android apps always perform blocking operations within the `doInBackGround` method in `AsyncTask`[2], such as network-send (Ln. 30). The message handler prepares the network-send parameter with the requester-supplied URL (Ln. 20). Once `execute` on the next line is called, the framework starts `doInBackGround` (Ln. 32) in a new thread, introducing another entry point to the component. Code that is reachable from each entry point is a segment of the entire component code. These segments can be statically determined via reachability analysis. We refer to them as *splits* (defined shortly). Although executing in separate contexts, splits are by no means isolated and in fact can relate to each other through inter-split data-flows. Heap and global variables used in different split can form these flows. Note that there exist two hijack-enabling flows that originate from the split started by `handleMessage` and reach to the split started by `doInBackground`: (i) the heap variable `currLoc` assigned with the location data (Ln. 15) and used as the `HTTP Post` content (Ln. 34), and (ii) the local array `sendParams` containing the URL (Ln. 20), implicitly passed to `params` on Ln. 32 by the framework as an entry point parameter, and used for the `HTTP Post` (Ln. 34). Therefore, our analyzer needs to be capable of:

C4 : Tracking data flows across splits and components.

In summary, this example demonstrates that a component is vulnerable to hijacks when it is exported to the public without limiting its interfaces to intended users. It also shows that using hijack-enabling data-flows to model the vulnerability is general

[2]A convenient threading construct provided by the framework.

and straightforward. A program analyzer aiming at detecting these flows faces four major challenges imposed by the unique Android programming paradigms (*C1, C2*) or by the complications of the data-flows (*C3, C4*). Next, we introduce our approach to conducting data-flow analysis on Android apps, with vulnerability detection as an application. We propose analysis methods and models that overcome the challenges discussed above. They are expected to be useful to other types of app analysis as well.

4.3 Analysis Methods and models

The reason why we chose to model the framework, instead of including all its code into the analysis scope, is because of the complexity of analyzing the framework code and the simplicity of modeling its external data-flow behavior. Due to the framework's extensive use of reflections, mixed use of programming languages, and overwhelming code size, including the framework code into the analysis scope incurs a significant amount of overhead and introduces certain extent of inaccuracy to the analysis. Therefore, analysis that only require a partial knowledge on the framework's external behavior, such as data-flow analysis, should model rather than diving into the framework, to avoid unnecessary performance overhead and inaccuracy. In addition to modeling the framework in terms of its data-flow behavior, our analysis requires type information of framework-defined classes (the app-level classes are derived from these types). We will show that, such information can be easily extracted from the framework, which the analyzer uses to build the complete class hierarchy.

4.3.1 Entry Point Discovery

As the first step to deal with the multi-entry-point nature of apps and tackle *C1*, we designed an algorithm that discovers entry points in app code at a very low false rate, without requiring analyzing the framework code. To avoid ambiguity, we use

Algorithm 1 Entry points discovery

$M_f \leftarrow$ {Uncalled framework methods overridden by app}
$M_a \leftarrow$ {App methods overriding framework}
$E \leftarrow$ {Listeners in Manifest; Basic component handlers}
repeat
  $G \leftarrow$ BuildCallGraph(E)
  for all $m_a \in M_a \wedge m_a$ overrides $m_f \in M_f$ **do**
    if m_a's constructor $\in G$ **then**
      $E \leftarrow E \cup \{m_a\}$
    end if
  end for
until E reaches a fixed point
output E as entry point set

the following definition of entry points in this paper:

Definition 2 *App entry points are the methods that are defined by the app and intended to be called only by the framework.*

Entry points in an app can be large in amount, often with a great variety in their object types. For instance, each UI elements in an app can define multiple event listeners to be called at different moments as particular events happen. Similarly, each component can implement handlers to get notified about its life-cycle changes. Therefore, we avoided any manual efforts that use expert knowledge to generate sets of possible entry points, due to its error pruning nature and no guarantee for completeness.

Since the entry point methods are supposed to be called by the framework, the latter then requires the prior knowledge about these methods. In fact, there are only two ways for an app to define entry points that can be recognized by the framework: either via explicitly stating them in the manifest file, or implicitly overriding methods or implementing interfaces that are originally declared by the framework as app entry points. Those defined using the first option can be determined by parsing the manifest. To find the rest, our algorithm first generates the set of uncalled methods in the app that override their counterparts declared in the framework, and then excludes

methods that are unreachable even by the framework (*i.e.* dead methods). Telling apart entry points from dead methods that override the framework, despite neither is called by the app, is based on two facts unique to entry points: (i) the containing class of any entry point always have at least one instantiated object (since app entry points are non-static methods), and (ii) there should be no app-level invocation on the original method that the entry point overrides or on any decedents of the original method in the class hierarchy. In contrast, dead methods that override the framework mostly cannot satisfy both conditions.

Our entry point discovery method, as formulated in Algorithm 1, follows an iterative procedure until a fixed point is reached for the entry point set E. Method set M_f and M_a are generated by a simple scan of the class hierarchy and all call sites in the app code. E is initialized to include entry points declared in manifest files and basic component-life-cycle handlers defined in the code. Compared with other entry points, the component-life-cycle handlers have very few types and are the only entry points whose containing class is created by the framework (*i.e.* calls to their constructors are invisible at the app level). During each iteration, a new call graph G is built based on the already discovered entry points in E. Due to the new entry points added in the last iteration, the new G may contain previously unreachable methods and classes instantiations. A method $m_a \in M_a$ is added to E as a new entry point when m_a overrides a framework method or interface $m_f \in M_f$ and m_a's containing class is instantiated in G. We build the call graph using the entire E, rather than just using the newly discovered entry points in the previous iteration, so that the point-to analysis supporting the call graph builder can be as complete and accurate as possible. The algorithm terminates when E stops growing and contains all possible entry points. Very rare false positives can happen only when framework-declared methods are never called in the app while they are already overwritten by instantiated classes and made for app use.

4.3.2 App Code Splitting

Once all entry points are discovered, we model their asynchronous invocations and addresses *C2* with a novel technique named *app code splitting*. We define the concept of splits as follows:

Definition 3 *A split is a subset of the app code that is reachable from a particular entry point method.*

From a static analysis perspective, app executions can be viewed as a collection of splits executing in all feasible orders, possibly interleaved. The idea of modeling app execution in terms of splits may seem challenging at the first glance. However, constrains imposed by the framework and our focus on data-flow analysis significantly simplify the realization of the idea. In fact, most splits in an app can only be executed in a sequential order (*i.e.* not interleaving each other), because the framework invokes the majority of app entry points in the main thread of an app. The mere exceptions are entry points of concurrency constructs, such as threads. Since our goal is to perform security vulnerability detection, concurrency-incurred data-flows are usually not a concern in this context due to their extreme unreliability to be reproduced or exploited. Therefore, we can safely approximate the app execution as sequential permutations of splits that are feasible under framework constraints.

Under this app splitting model, our data-flow analysis first computes the *split data-flow summary* (SDS) for each split in the app. It then starts the permutation process and, for each possible sequence of splits, generates *permutation data-flow summary* (PDS) by linking the SDS of each split in the sequence. As the permutation proceeds, each PDS is checked for *interesting data-flows* specified by pre-defined *policies*. Eventually, all possible data-flows can happen in the app are enumerated.

Figure 13 shows two SDS marked by dashed boxes. They are generated based on the two entry points, `handleMessage` and `doInBackground`. An SDS consists

of intra-split data-flows whose end nodes represent: (i) heap variables[3] entering or exiting the split scope (depicted by octagons); or (ii) pre-defined sources or sinks (depicted by rectangles). We omitted intermediate nodes in the SDS in Figure 13 to ease illustration. In essence, an SDS only contains data-flows within a split that may contribute to connecting a source to a sink (may resides in another split). We refer to these data-flows as *interesting-flows* hereafter. The upper SDS in Figure 13 has two isolated data-flows: the one on the left propagates the location data (a sensitive source, tagged as `Tag_SensSrc`) to a heap variable `currLoc`, and the one on the right carries the requester's input (tagged as `Tag_InputSrc`) to a transit sink; The lower SDS captures the convergence of a heap variable and a transit source at a sink associated with two tags (`Tag_DataSink` and `Tag_CriticalSink`, explained shortly). We compute SDS via a context- and field-sensitive data dependence analysis, identifying interesting-flows in the current split. As Figure 13 shows, heap variables are represented by their heap location key, which is a three-tuple in the form of $(field, allocSite, method)$, indicating the $field$ whose containing object was allocated at $allocSite$ in $method$ ($field$ of any array object is `null`). Pre-defined sources and sinks (data entry or exit points of the analysis's interest) are represented by a four-tuple, $(method, paramIndex, tag, callSite)$, indicating that a parameter of a method called at $callSite$ is a source or sink depending on the tag.

Our analysis method allows for a fairly flexible way of defining and extending tags associated with sources and sinks. Tags are used to differentiate sources and sinks with different semantic meanings given by the analyzer users based on their specific usage scenarios. Policies that specify interesting-flows can be defined based on the tags associated with their end nodes:

$P := F_{int} \bowtie [F_{int} \mid \emptyset]^n, \quad F_{int} := [Tag] \rightsquigarrow [Tag],$ where $\bowtie$ defines a *join* relationship exists between two interesting-flows (*i.e.* two flows, or their extensions,

[3]Variables with a global scope, as opposed to local variables.

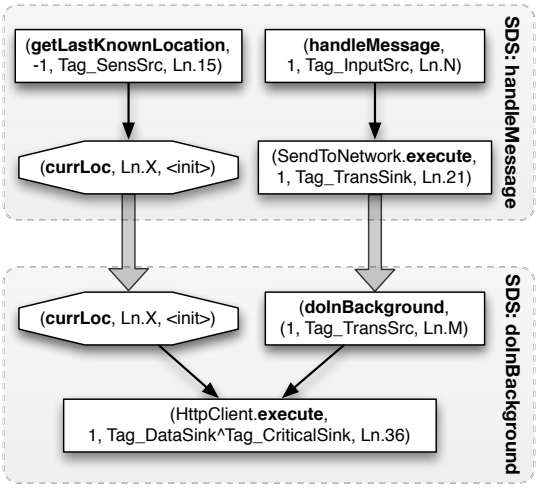


Figure 13: Linked-SDS for the running example

intersect or converge with each other), and $\rightsquigarrow$ defines an interesting flow with two end nodes of specified tags. By supporting customizable tags and the *join* relationship in defining interesting-flows, our analyzer provides a means of expressing the side-effects of converged flows on a semantic level, which solves *C3*.

For component hijacking vulnerability detection, we define two general source tags, `Tag_SensSrc` and `Tag_InputSrc`, to mark the start points of interesting-flows that propagate sensitive information or requester's input. We also define three general sinks to mark end points of interesting-flows that are to make data publicly accessible (`Tag_PublicSink`), make data accessible to specified entities (`Tag_SpecifiedSink`), or write data into critical data regions (`Tag_CriticalSink`). With these tags defined, we can easily convert Definition 1 into three simple policies to capture hijack-enabling flows:

P1 : $\{\texttt{Tag\_SensSrc} \rightsquigarrow \texttt{Tag\_PublicSink}\}$

P2 :

$\{\texttt{Tag\_InputSrc} \rightsquigarrow \texttt{Tag\_CriticalSink} \bowtie \texttt{Tag\_SensSrc} \rightsquigarrow \texttt{Tag\_SpecifiedSink}\}$

P3 : $\{\texttt{Tag\_InputSrc} \rightsquigarrow \texttt{Tag\_CriticalSink}\}$

These policies are checked on every newly generated PDS as the split permutation continues. For the example, our analyzer can detect the hijack-enabling flows, satisfying *P2* and *P3*, from a PDS that links the SDS of *handleMessage* with that of *doInBackGround*, as shown in Figure 13.

The PDS generation is carried out by two basic operations – link and unlink an SDS. The link operation adds a new SDS into a PDS if inter-split data-flows exist from the latter to the former. It draws data-flow edges (*e.g.* the two thick edges in Figure 13) from leaf nodes in the PDS to those reachable root nodes in the new SDS. For Android apps, the only two channels through which data can flow across splits are: heap variables sharing the same location key tuple, and framework API pairs that transit data among splits. We introduce a pair of special tags, `Tag_TransSink` and `Tag_TransSrc`, to model these API pairs. The link operation can reject the SDS if no edge can be drawn and the SDS does not contain flows starting with any predefined source. A rejection suggests that the new SDS has no effect on any potential propagation of interesting-flows in the current PDS, and thus, there is no need to add it. Unlink operation simply reverts the last link operation.

Intuitively iterating through all split permutations can be a prohibitively expensive operation for apps with a large number of entry points. We leverage on the continuity of data-flows across splits to carry out a simple but effective search pruning. The depth-first search only appends an SDS to the current permutation if it is accepted by the link operation and then continues iterating along that path. As shown in Section 4.5, this pruning greatly reduces the search space and time overhead of the permutation process. The permutation also considers a few constraints on the launch order of splits that handle life-cycle events of basic components (*e.g.* entry points relating to component initialization and termination are called in fixed orders).

Finally, *C4* is addressed, because all interesting-flows in an app, both intra-split and inter-split ones, are constructed during the split permutation process. Our app

splitting technique enables a data-flow analysis that is more efficient and better accommodates the event-driven nature of Android apps, than the conventional methods, which synthesize a main function explicitly invoking event handlers. App splitting creates a divide-and-conquer theme. The sub-problems (*i.e.* constructing intra-split data-flows and SDS) are significantly easier and smaller in scale than the original problem (*i.e.* constructing data-flows for an entire app, as faced in the conventional methods). The merge process (*i.e.* permuting splits) can be very fast. Moreover, due the mutual independence among SDS, they can be built in parallel and cached for reuse (*e.g.* SDS for common libraries can be built once and reused when analyzing all apps that make use of them) to further improve the performance.

4.4 Implementation of Dalysis and CHEX

We built a generic Android app analysis framework named Dalysis, which stands for Dalvik bytecode analysis. As suggested by its name, Dalysis directly works on off-the-shelf app packages (or Dalvik bytecode) without requiring source code access or any decompilation assistance. Previous app analysis efforts that relied on decompiled source code have two major drawbacks — heavy performance overhead and incomplete code coverage. As reported by Enck *et al.* [35], the state of the art technique to decompile an app, on average, takes about 27 minutes and leaves 5.56% of the source code failed to be recovered. Conducting analysis at the dalvik bytecode level overcomes these issues. In addition, unlike x86 binary code, bytecode retains sufficient program information from the high level language and does not have any parsing ambiguity, thus serves as an ideal analysis subject.

To our best knowledge, Dalysis is the first generic analysis framework that operates on Dalvik bytecode and intended to support multiple types of program analysis tasks. Next, we introduce the internals of Dalysis that can facilitate the understanding of the implementation of CHEX, our component hijacking analyzer built based on Dalysis.

We leave out the low-level system building details as they are out of the scope of this paper.

4.4.1 Dalysis Framework

The front end of Dalysis consumes an Android app package (`.apk`) at a time. It retrieves package information from metadata files and translates the Dalvik bytecode into an intermediate representation (IR), based on which the back end analyzers carry out their tasks. The front end starts the IR generation process by parsing the input bytecode file. Dalysis employs an open source Dalvik bytecode parser named `DexLib`, part of a well-known disassembler for Android apps [3]. `DexLib` provides useful interfaces to programmatically read embedded data, type information, and Dalvik instructions from a bytecode file. Dalysis allows different analysis to choose either include the entire Android framework code or model its external behaviors, which is achieved by linking two different versions of the runtime library into the analysis scope. The front end constructs the class hierarchy, performs an semantical IR translation from Dalvik and Java bytecode (Android framework libraries are compiled into java bytecode), and then hands over the IR to backend analyzers.

We adopted our IR from the `WALA` project [7], a popular static analysis framework for Java, for two reasons: the semantic proximity between Dalvik bytecode and the IR and a wide selection of basic analyzers developed for the IR by the `WALA` community. The translation process is mostly straightforward, since both instruction sets follow the register-machine model and retain a similar amount of information from the same high level language (*i.e.* Java). However, a handful of instructions that are unique to Dalvik virtual machine require special handling during the translation process. For example, the `filled-new-array` instruction allocates and initializes an array in one step; And the `move-result` instruction retrieves the result of the previous call from the special `result-register`. Following the semantic translation is the final

task for the front end – static single assignment (SSA) conversion. The conversion performs an abstract interpretation on each method, wherein the define-use chain is determined for each Dalvik register as well as its mapping to the local variable on Java level. New instructions are generated, as a side effect incurred by the flow function of the abstract interpretation. As a result of variable renaming (*i.e.* a register model conversion), newly generated instructions operate on a conceptual register model with unlimited registers, each of which can only be assigned once as required by SSA form. `Meet` operations happen at basic block boundaries. As a result, ϕ variables are generated to merge two or more values that may flow into a same variable in the current basic block from predecessors in the control flow graph. Converting the IR into SSA form can simplify various types of program analysis, especially data-flow related ones, such as definition reachability test, constant propagation and etc. In fact, many existing analyzers for `WALA` assume an SSA IR.

The back end of Dalysis hosts a variety of analyzers and provides them the interfaces to access the IR, the class hierarchy, and other useful information. Some basic analyzers released by `WALA`, such as the point-to analysis and the call graph builders, are included in Dalysis. These building-block analyzers can be found useful by many advanced analyzers. Dalysis itself is not specific to any particular flavor of app analysis — it is designed to be a generic framework that can enable as many types of analysis as possible on Android apps. For example, CHEX demonstrates how we implemented the data-flow analysis methods, introduced in Section 4.3, by using the Dalysis framework.

Dalysis is implemented in Java with 15,897 lines of source code, excluding 3rd party libraries. The building process took us a significant amount of efforts due to a lack of similar work and reusable code. But most efforts were spent on tackling engineering related issues or implementing existing algorithms from the programming language community, therefore we do not intend to claim these efforts as contributions

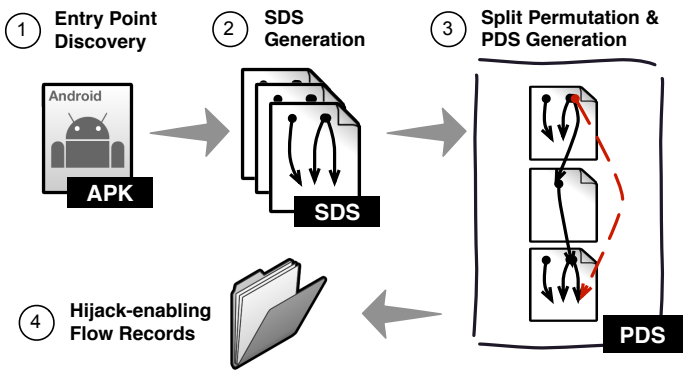


Figure 14: CHEX workflow

in this paper. We also omit the implementation details of Dalysis that should be oblivious to analyzer designers, which is out of the scope of this paper.

4.4.2 CHEX: Component Hijacking EXaminer

CHEX realizes our data-flow analysis methods and models discussed in Section 4.3. It detects hijack-enabling flows based on policies *P1-3*, with a set of 180 sources and sinks that match the tags defined by these two policies. This set was constructed semi-automatically to cover a relatively wide range of hijack-enabling flows that affect the sensitive resources managed by the system (*i.e.* protected by Android permissions and accessed uniformly across apps). Parts of the sensitive sources (`Tag_SensSrc`) were selected based on the API-to-permission mapping provided by [38]. This set is adequate for our testing and evaluation purpose, but it is not meant to be complete. In fact, it can be extended with source and sinks specific to individual apps, so that CHEX can capture hijack-enabling flows in app's semantics.

As shown in Figure 14, entry point discovery starts at first. It queries Dalysis front-end for information necessary to the initialization process, such as event listeners defined in manifest and method overloading relationships (shown in Algorithm 1). CHEX makes multiple different uses of the call graph builder from `WALA`, which can be configured to have different degrees of context sensitivity. For each iteration in

the entry point discovery process, we generate a context-insensitive call graph, for the least performance overhead and the unnecessity of context sensitivity in this scenario (*i.e.* we use the call graph only to conservatively estimate if a method was called or a class was instantiated before).

For each discovered entry point, or more specifically, the split started by that entry point, CHEX builds an SDS to summarize its data-flow behaviors that may contribute to forming any hijack-enabling flow (Step 2 in Figure 14). Building SDS is a computation heavy step in the entire analysis because it is where all intra-split data-flows are constructed directly by analyzing the IR. In comparison, in a later step, the permuter generates inter-split flows and PDS based on simple rules determining the connectivity between two intra-split flows.

Conventional data-flow analysis approaches solve data-flow equations through an iterative process. This process is expected to reach a fix-point after limited iterations of basic-block state changes made by transfer functions. However, for the purpose of building SDS, we can safely avoid this procedure and still be able to check interesting-flows, thanks to the SSA IR and our abstraction of the flow checking problem. Specifically, the SSA conversion carried out by the front end has already conducted a basic data-flow analysis and saved information (*e.g.* variable use-define chains and *etc.*) that can greatly facilitate the construction of system dependence graphs. Inspired by the way of utilizing system dependence graphs in the classic program slicing algorithm [49], we convert the problem of checking interesting data-flows into an equivalent graph reachability test problem. We test the connectivity of source-sink pairs on customized system dependence graphs that only have data-dependence edges (referred as *data-dependence graph*, or DDG). A source-sink pair that is connected on a DDG indicates the existence of a data-flow from the source to the sink. Compared with the conventional approaches, this abstraction offers us a better leverage on the existing IR and avoids unnecessary analysis work, yet still

achieving the same goal.

DDG is constructed in a similar way as system dependence graph is in [49], but without generating control-dependence edges. Each node in DDG represents either a normal SSA statement or an artificial statement to model inter-procedure parameter passage. An edge is drawn from node S_1 to node S_2 only when the variable defined by S_1 is directly used by S_2. Intra-procedural edges between scalar variables are drawn with the help of local use-define chains implied from the SSA IR. Identifying inter-procedural dependencies among heap variables requires a call graph with a proper degree of context sensitivity and an inter-procedural definition reachability analysis. We chose a `0-1-CFA` call graph builder with the `call-string` context sensitivity (*i.e.* using the calling string to identify a particular node in the call graph), for its sufficient accuracy and acceptable performance overhead. With the call graph, regular parameter and return passing edges can be added between the corresponding callers and callees. The definition reachability analysis provides information about (transitive) heap variable accesses in a method, which is needed to create heap related nodes and draw edges between them (inter-procedural heap variable accesses are modeled as artificial parameters or returns).

Before used for the interesting-flow discovery, a DDG needs to go through an edge inflation process, as a way to model data dependencies that are still missing. Missing edges are resulted from out-of-scope code (*i.e.* methods defined outside of the analysis scope). Thus we need to model the external data-flow behavior of such code. The modeling can be easily done by means of adding artificial edges into the DDG, bases on two simple rules: (i) for methods with returns, the return value is dependent on all parameters (*i.e.* drawing edges from each `ParameterCaller` node to the `ReturnCaller` node); and (ii) for return-less methods, the first parameter (*i.e.* `this*` for non-static methods) is dependent on all other parameters, if any (*i.e.* drawing

edges to the define node of the first parameter from other `ParameterCaller`). Exceptions to these rules do exist, but only very few happen frequently enough that we need to specially handle, such as several methods of strings and collection types.

With the DDG is generated, searching for interesting flows becomes intuitive. CHEX first picks two sets of nodes from the graph, S_{start} and S_{end}, where S_{start} contains pre-defined sources (*i.e.* start points of inter-split flows), and S_{end} contains pre-defined sinks (*i.e.* end points of inter-split flows). CHEX then constructs the SDS as it traverses the DDG – a flow is added to the SDS if it starts from a node in S_{start} and ends with a node in S_{end}. The resulting SDS serves as a gadget for the permuter to compute PDS (Step 3 in Figure 14). Although the SDS building process is the most computation-intensive step during the entire analysis, the problem size is already greatly reduced, comparing with conducting the similar analysis on the whole app without app splitting. Tasks performed during the SDS construction, such as point-to analysis, generally scale poorly as the app size increases. Therefore, dividing the app into smaller but self-contained splits can help with the performance, and alleviates the scalability issue for large apps. In addition, due to their independence, SDS constructions for different splits can be carried out in parallel in performance-critical and computing-resource-rich scenarios, to further reduce the overhead.

The split permuter always starts a new sequence with a split from an exported component, a constraint to reflect the causal relationship between external requests and potential hijack-enabling flows. The permutation is implemented as a regular depth-first-search with pruning and configurable search space. For example, the maximum DFS depth specifies the maximum number of splits a feasible hijack-enabling flow can span through, a practical trade-off between performance and completeness. As the permutation proceeds, interesting flows in the current PDS are matched with policies *P1-P3* for hijack-enabling flows. Node tags and the $\rightsquigarrow$ relation can be simply

checked on individual interesting flows. As for the $\bowtie$ relationship, we test if two interesting flows merge into a new variable or join at a same method call site. Discovered hijack-enabling flows are recorded (Step 4 in Figure 14) with detailed information, such as the corresponding paths in PDS, the split sequence, and the policy they satisfy. Such information can assist app developers or security researchers to verify and fix vulnerabilities.

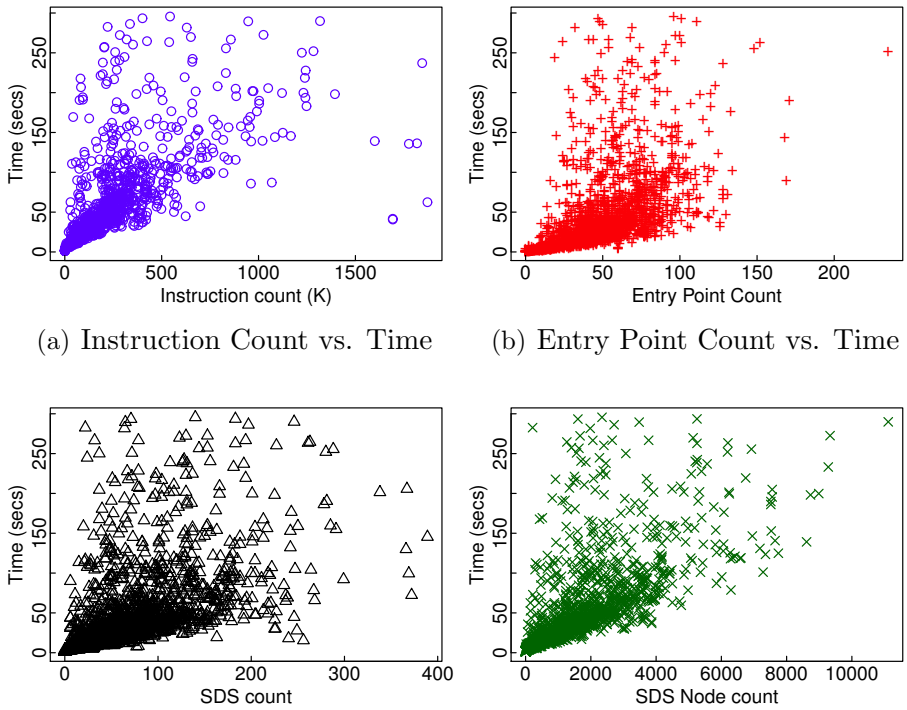
4.5 System Evaluation and Experiments

We exercised CHEX with a large set of real-world apps, S_{pop}, containing about 5,486 free popular apps we collected in late 2011. S_{pop} consists of around 3,486 apps from the official Android market and 2,000 from alternative markets. The experiments were conducted on a cluster of three computers, each equipped with an Intel Core i7-970 CPU and 12GB of RAM. During the experiments, we launch concurrent CHEX instances on 64-bit JVM with a maximum heap space of 4GB. To optimize the throughput, we limit the processing time of each app within 5 minutes.

4.5.1 Performance

We instrumented CHEX to measure its execution time while it examining apps in S_{pop}. The median processing time for an app is 37.02 seconds with the interquartile range (IQR) of 161.87 seconds, which suggests that CHEX can quickly vet a large amount of apps for component hijacking vulnerabilities. 22% apps needed more than 5 minutes to be analyzed thus timed out in our experiments. In practice, with more computing resources available, a more generous time-out value should be used.

We found that CHEX's execution time varies significantly across different apps. As a result, we studied the impact of four app-specific factors that may affect CHEX's execution time the most (see Figure 15). Although these factors are in a strong correlation with the execution time, no single factor dominates it (*i.e.* none poses major bottleneck to the performance). Furthermore, we decomposed the execution time



(a) Instruction Count vs. Time (b) Entry Point Count vs. Time

(c) SDS Count vs. Time (d) SDS Node Count vs. Time

Figure 15: Timing Characteristics of the Analysis

into three parts, corresponding to the three analysis phases each app goes through, as shown in Figure 16). In general, SDS construction (or split permutation) causes the majority of the time overhead, whereas entry point discovery and DFS generation often finish fast.

Some findings acquired during the evaluation also prove that the app analysis challenges we tackled in this work ($C1 - C4$) are very common to encounter when analyzing real apps. On average, we found 50.37 entry points of 44 unique class types in an app. Moreover, the number of entry points is not directly related to the app size. Apps implementing complex user interfaces or requiring frequent user interactions (*e.g.* games) tend to have more entry points than others. About 99.70% of apps contain inter-split data flows, which strongly indicates the necessity of analyzing such flows the contexts created by different entry points.

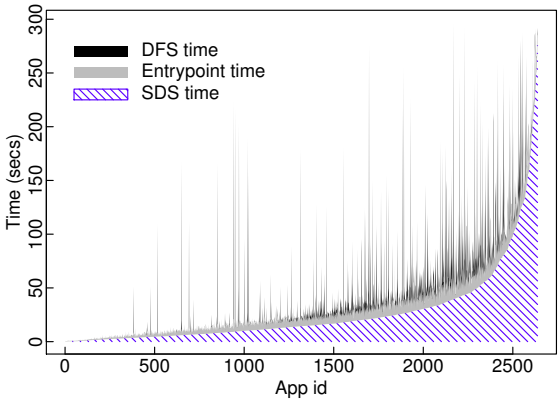


Figure 16: Performance Decomposition

4.5.2 Accuracy

Among the 5,486 apps in S_{pop}, CHEX flagged 254 as potentially vulnerable to component hijacking attacks. Due to the lack of a ground truth, we manually verified all the flagged apps by checking if the discovered hijack-enabling flows are indeed feasible and exploitable by attackers. This verification process largely replied on human expert knowledge with the assistance of well-known Android app disassemblers and decompilers. In the end, we identified 48 flagged apps as false positives, which yields a true positive rate above 81%. The main causes for the false positives are infeasible split permutations and apps' complicated input validations that CHEX cannot understand. Although the false positive rate is acceptable in a vulnerability filtering scenario, we argue that the first cause can be minimized by incorporating Android domain knowledge into the permutation pruning logic, while the second cause is a difficult but orthogonal issue to this work (*i.e.* checking the quality of program's input validation).

4.5.3 Case Studies

Our manual verification process also helped us gain practical insights into the component hijacking vulnerabilities. All 206 apps that are confirmed as vulnerable can be roughly categorized into five classes, as shown in the first column of Table **??**. It

clearly shows that, in addition to vulnerabilities exploited by confused deputy attacks on the Android permission system, other vulnerability classes also fall into the scope of component hijacking and can be detected by CHEX. The second column refers to Definition 1 and indicates the hijacking type for each vulnerability class. To improve the community's awareness and understanding of component hijacking, we selected at least one app from each class and conducted the following case studies. We hide part of the app package names as a precaution to not leak undisclosed vulnerability information.

Case A1 in the data theft class resembles the example app we used in Figure 12. One of its components obtains the GPS location and saves it to a global variable. Another component initializes a URL parameter using a string provided by an arbitrary app via Intent, and sends the GPS information to the URL. An attacker thus can steal the sensitive location information by sending a crafted Intent to the second component, causing the GPS location to be sent to the attacker controlled server.

Apps can also leak their private, permission-protected capabilities through public components, as previously reported. Case B1 has a public component that takes a string from another app's Intent and uses it as a URL for Internet connection. Likewise, a public component of case B2 uses a string from an Intent as the host name for socket connections. These vulnerable apps essentially give out the Internet permission to all other apps who may not have it. For example, a malicious app can exploit these apps to transmit information to an arbitrary Internet server, or even launch network attacks against a victim server. We have observed Internet capability leakages in both Activity and Service components of vulnerable apps. In the cases of Activity components, the exploited components can be forced to display specified remote content to the user; Whereas exploits on vulnerable Service components can be carried out more stealthily, because Service components execute in the background (in this case, communicating with attacker controlled servers) without interacting

with users.

Intent proxy is another class of vulnerabilities that can be exploited in a fashion similar to capability leak. Case C1 accepts an input Intent (X) that embeds another Intent (Y). It then starts a new Activity per Y's request using its own identity. More specifically, in the `OnResume()` function of C1, the Intent Y is retrieved from the `Bundle` object through the key "intent". Next, Intent Y is directly passed to `startActivity` without checking any properties of Intent Y. With this proxy, an attacking app can hide its identity and start activities, even those protected by permissions that C1 has but the attacking app does not.

Android heavily relies on internal SQL database to organize system and app data, such as contacts and app private information. Apps can interact with its database using APIs that take SQL statements as arguments. Case D1 passes an input string from an Intent directly into a raw SQL query, which allows attackers to inject SQL statements to manipulate the database or even cause system comprises. In addition, we have also uncovered more subtle SQL injection vulnerabilities in many apps, which use parameterized query instead of raw query, but in a non-parametric form. In particular, the vulnerable apps construct the selection clause of a query by directly inserting unescaped strings from Intent, instead of passing them in a parameter array. Such practices allows the attackers to inject an arbitrary condition into the selection clause, and derail the execution of the query, causing unexpected behaviors of the victim app. Besides SQL injections, a similar but more harmful vulnerability, as in Case D2, is the shell command injection, where app issues Linux shell commands using unchecked input strings.

The last class, data tampering, leads to private or critical data being overwritten by attackers. Case E1 is a game that reports user's score to a remote server for ranking purposes. However, the reporting component is made public and reports arbitrary scores specified by a requestor, which creates an easy way for cheating the

game's online scoreboard. We also observed a more security-critical case where the payment URL of an online shopping app can be modified by attackers. The extent of damage by this type of vulnerability is highly dependent on the function of individual apps, as well as the robustness of client-server interactions of the apps.

4.6 Discussions

As the evaluation shows, CHEX do have false positives. However, they can be reduced by addressing two limitations of our current prototype. First, our prototype does not leverage on much domain knowledge about the partial orders in which Android components and their entry points can run or interleave. This design choice was made because PDS construction enforces the data-flow continuity between splits, which sorts out the majority of infeasible split permutations but not all. In addition, building such domain knowledge, possibly time-consuming and error-prune, is out of this work's scope. We argue that when adopted in practice, CHEX can always incorporate new constraints into the split permeation, which not only reduce the false positive rate but also improve the performance. Second, our current prototype is unable to recognize false hijack-enabling flows that are sanitized by complicated logic (*e.g.* regular expression matching and etc.), because it by itself is an open research problem. On the other hand, we observe that the majority of apps rely on simple framework APIs (*e.g.* `checkCallingPermission`) and constant string matching to carry out effective input validation, which are already handled by CHEX.

The fact that CHEX only checks data-flows to detect vulnerabilities may cause false negatives. Rare vulnerable components may exist that enable hijacking attacks without explicit data-flows. In these cases, date dependencies are essentially encoded into control dependencies and thus sources and sinks are no longer connected via data-flows. We could selectively track control dependence for certain sources

(*e.g.* `Tag_InputSrc`) in our SDS, so that implicit intra-split data-flows can be considered during analysis. On the other hand, control dependency analysis can also easily bring false positives. The study of this trade-off is out of the scope of this paper.

4.7 Related Work

Event-driven (callback-based) programming is widely used in implementing graphical user interface (GUI) and web systems. To statically analyze GUI systems, previous work [78, 77] leverage on domain knowledge to identify and to configure the entry point (callback) methods. In web systems, event handler functions are easy to identify given the uniform ways to define them. However, in Android, the large number of entry point types makes it difficult to identify them completely—previous work relied on specific domain knowledge to detect common component entry points without guarantee for completeness [41]. We devise a heuristic-based approach to discover all possible entry points to the apps with low false positives. To model the execution of multiple entry points, previous work [78, 77] employ a synthetic main function to mimic the event loop dispatcher in GUI systems. We introduce SDS to summarize intra-split data-flows and permute the splits to model their asynchronous invocations and derive the inter-split data-flow behaviors. Comparing with [78, 77], we divided the global data-flow analysis problem into much smaller but self-contained sub-problems, which improves the performance and scalability.

Static analysis and model checking have a history in assisting vulnerability discoveries [26, 56, 53, 37]. For web systems, Jovanovic *et al.* designed Pixy [53] to detect input validation flaws in server side scripts written in PHP through an interprocedural context-sensitive data flow analysis. A similar study has been carried for cross site scripting vulnerabilities [86]. Bandhakavi *et al.* applied a context-sensitive and flow-sensitive static analysis for analyzing the security vulnerabilities of Firefox

plugins written in JavaScripts [22]. For Java programs, Livshits *et al.* designed a datalog language to describe the security policies that direct vulnerability detection [56]. Tripp *et al.* built an industrial strength static taint analysis tool [81]. Comparing with the aforementioned efforts, we focused on detecting component hijacking vulnerabilities in Android apps. We first tackled general challenges faced by static app analyzers due to Android's special programming paradigm, and then proposed a data-flow-based detection approach.

Security mechanism based on information flows, such as JIF [64], HiStar [89] and Asbestos [31], are also related in that our work define and detect component hijacking by means of data-flow policies, despite that we do not enforce the policies in runtime.

Mobile security issues have gained much attention recently. Malware are not strangers for both the official Android market and alternative ones [91]. Research efforts were made on detecting repackaged apps [90] or apps with known malicious behavior [92, 41]. Recently Google also launched its malware filtering engine [2]. Information leakage is another major security threat for mobile devices. Kirin [36] detects apps whose permissions might indicate potential leakage. TaintDroid [34] leverages dynamic taint analysis to detect information leakage at runtime. PiOS [33] addressed the same problem using static analysis for iPhone app. In general, information leakage detection reveals the potential out bound propagation of sensitive information, which might be benign in many cases. Instead, component hijacking detection captures the information leakages resulted from an exploitation (*i.e.* sensitive data theft), in addition to other hijacking types.

Enck *et al.* introduced Ded [35] to convert Dalvik bytecode back to Java bytecode, and then used existing decompilers to obtain the source code of the apps for analysis. Our Dalysis framework directly converts Dalvik byte code to an SSA IR and enables various types of static analysis. Unlike the decompilation process, our IR conversion is sound (*e.g.* no heuristics or failures) and costs much less time. We model the Android

framework and its special program paradigm rather than coarsely treating apps as traditional Java programs. As a result, our analysis is more tailored for Android apps and thus has better precision.

Android mediates access to protected resources using a permission system. However, its effectiveness hinges on app developers correctly implementing it. Chin *et al.* showed that apps may be exploitable when servicing external intents [28]. They built ComDroid to identify publicly exported components and warn developers about the potential threats. For this purpose, it is sufficient for ComDroid to only check app metadata and specific API usages, rather than performing an in-depth program analysis as CHEX does. As a result, warned public components are not necessarily exploitable or harmful (*i.e.* the openness can be by design or the component is not security critical). On the other hand, Android permission system is subject to several instances of the classic confused deputy attack [46]. As demonstrated by [29, 55, 39, 41], an unprivileged malicious app can access permission-protected resources through privileged agents (or app components) that do not properly enforce permission checks. Recently proposed runtime mitigations either reduce the agent's effective permissions to that of the original requestor [39] or inspect the IPC chains for implicit permission escalations [30, 24]. While these runtime solutions are effective at protecting end users adopting them, scalable detection methods for the problematic agents in question (*i.e.* hijack-able components) are still important to have in order to prevent vulnerable apps from reaching the vast users in the first place. Grace *et al.* [41] employed an intra-procedural path-sensitive static analysis to discover permission leaks specific to stock apps from multiple device vendors. In comparison, CHEX targets at a more general vulnerability in all types of Android apps and performs inter-procedural analysis with high degrees of sensitivity. Thanks to our novel entry point discovery and app-splitting techniques, CHEX is capable of accommodating Android's special programming paradigm and finding complex hijack-enabling

flows. It is also noteworthy that the component hijacking attacks we address includes but is not limited to attacks targeting at permission-protected resources.

4.8 Conclusion

We defined and studied the component hijacking problem, a general category of vulnerabilities found in Android apps. By modeling the vulnerabilities from a data-flow perspective, we designed a static analyzer, CHEX, to detect hijack-enabling data-flows in a large volume of apps. In doing so, we introduced our method to automatically discover entry points in Android app, as well as the novel analysis technique, app splitting, as an efficient and accurate way to model executions of multiple entry points and facilitate global data-flow analysis. We also built the Dalysis framework to support various types of static analysis directly performed on Android bytecode. CHEX prototype was implemented based on Dalysis and was evaluated with 5,486 real-world apps. The empirical experiment demonstrated a satisfactory scalability and performance of our analysis method, as well as provided an insight into the real-world vulnerable apps we detected.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

My thesis as a whole explores, realizes, and evaluates the new perspective of securing software and system, which limits or avoids the unwanted security consequences caused by unwary users. My work shows that, with the proposed approaches, software can be reasonably well protected against the attacks targeting the unwary users, which have been emerging into the major threats to today's cyber systems inside enterprises, governments, and other organizations. This new perspective brings an unconventional thinking into the research of software and systems security—instead of focusing on the attack specifics and their individual mitigations, security monitors for certain threats should center around the users, including their interactions with software, and the unexpected consequences. Because the monitors designed in this way provide generic and robust coverages. As demonstrated by BLADE and SURF, the sophisticated threats targeting users tend to exhibit invariant characteristics when being examined once users' intents or the impact of their actions are understood. Therefore, detection methods leveraging such understandings can remain attack-agnostic and achieve more complete and accurate results than previous ones.

Moreover, the knowledge and insights gained throughout the course of developing the thesis have advanced the community's understanding of several emerging threats and related problems: SURF provided the first large-scale study of search poisoning campaigns in the wild and revealed their sophisticated operations; CHEX found previous unknown vulnerability classes in Android apps and generalized similar vulnerabilities into the general component hijacks. As a result, they have inspired and

guided many follow up research works. More importantly, these insights have emphasized the increasing importance of considering the wide existence of unwary users when designing and securing systems.

Each work included in this thesis has yielded at least one practical threat mitigation system, including several that have been adopted in real-world products and services. Evaluated by the large-scale real-world experiments, these systems have demonstrated their effectiveness at thwarting the security threats faced by most unwary users today. The threats addressed by this thesis have span multiple computing platforms, such as desktop operating systems, the Web, and smartphone devices, which highlight the broad impact of the thesis.

In terms of the possible extensions of this thesis and future research, I vision that the fast growing popularity and the privacy-bearing nature of mobile devices will spur powerful and advanced attacks targeting at unwary users, vulnerable apps, and OS design flaws. Therefore, more research on comprehensive app vetting methods are needed to aid the underdeveloped field of app quality assurance, and in turn reduce the amount of vulnerable apps arriving at end users in the first place. In particular, it is very useful to design mobile app analysis methods that can generically identify design flaws leading to privacy leak or resource abuse. This is because such high-level flaws threaten mobile users more significantly than the hard-to-exploit implementation errors on mobile platforms. Improving design-in security in mobile OS is also an important research direction. Although mobile operating systems have inherited the-state-of-the-art security mechanisms from their desktop counterparts, their lack of considerations of the mobile utilities and the user factors requires revisits of the OS security design. Design-in security on mobile platforms is expected to, among other goals, contain the unwanted security impact of uninformed or misinformed users.

REFERENCES

[1] "Alexa - Top Sites By Category." http://www.alexa.com/topsites/category.

[2] "Android and security." `http://googlemobile.blogspot.com/2012/02/android-and-security.html`.

[3] "Baksmali: a disassembler for Android's dex format." `http://code.google.com/p/smali/`.

[4] "Google trends." http://www.google.com/trends.

[5] "Quality of Android market apps is pathetically low." `http://www.huffingtonpost.com/2011/06/20/android-market-quality_n_880478.html`.

[6] "URLVoid: Scan a website with multiple scanning engines." http://www.urlvoid.com/.

[7] "WALA: T.J. Watson libraries for analysis." `http://wala.sourceforge.netl`.

[8] "WOT: Web of trust." http://www.mywot.com/wiki/API.

[9] *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, 1993.

[10] "Malware poisoning results for innocent searches." http://www.eweek.com/c/a/Security/Malware-Poisoning-Results-for-Innocent-Searches/, November 2007.

[11] "finjan: securing your web." http://www.finjan.com, 2009.

[12] "stopbadware.org." http://www.stopbadware.org, 2009.

[13] "Symantec inc." http://www.symantec.com, 2009.

[14] "Barracuda labs 2010 mid-year security report," tech. rep., Barracuda Networks Inc., 2010.

[15] "Operation aurora." `http://en.wikipedia.org/wiki/Operation_Aurora`, April 2010.

[16] "Search engine optimization 'poisoning' way up this year." http://www.networkworld.com/news/2010/110910-seo-poisoning-increases.html, November 2010.

[17] "Google: Search engine spam on the rise." http://www.pcworld.com/article/217370/google_search, January 2011.

[18] "RSA: SecurID attack was phishing via an excel spreadsheet." http://threatpost.com/en_us/blogs/rsa-securid-attack-was-phishing-excel-spreadsheet-040111, April 2011.

[19] "Android application components." http://developer.android.com/guide/topics/fundamentals.html#Components, 2012.

[20] "Espionage hackers target watering hole sites." http://krebsonsecurity.com/2012/09/espionage-hackers-target-watering-hole-sites/, September 2012.

[21] Adam, B., Adrienne, P. F., Prateek, S., and Aaron, B., "Protecting browsers from extension vulnerabilities," in *Network and Distributed System Security Symposium (NDSS)*, 2010.

[22] Bandhakavi, S., King, S. T., Madhusudan, P., and Winslett, M., "Vex: vetting browser extensions for security vulnerabilities," in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[23] Barth, A., Jackson, C., Reis, C., and Team, T. G. C., "The Security Architecture of the Chromium Browser," in *Stanford Technical Report*, 2008.

[24] Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A.-R., "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Tech. Rep. TR-2011-04, Technische Universitat Darmstadt, 2011.

[25] Carbone, M., Cui, W., Lu, L., Lee, W., Peinado, M., and Jiang, X., "Mapping kernel objects to enable systematic integrity checking," in *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, (New York, NY, USA), pp. 555–565, ACM, 2009.

[26] Chen, H. and Wagner, D., "Mops: an infrastructure for examining security properties of software," in *Proceedings of the 9th ACM CCS*, 2002.

[27] Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and Wang, Y.-M., "A systematic approach to uncover security flaws in gui logic," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2007.

[28] Chin, E., Felt, A. P., Greenwood, K., and Wagner, D., "Analyzing inter-application communication in android," in *Proceedings of the 9th MobiSys*, 2011.

[29] Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M., "Privilege escalation attacks on android," in *Proceedings of the 13th ISC*, 2010.

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

[31] Efstathopoulos, P., Krohn, M., VanDeBogart, S., Frey, C., Ziegler, D., Kohler, E., Mazières, D., Kaashoek, F., and Morris, R., "Labels and event processes in the asbestos operating system," in *Proceedings of the 20th ACM SOSP*, 2005.

[32] Egele, M., Wurzinger, P., Kruegel, C., and Kirda, E., "Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks," in *Proceedings of Detection of Intrusions and Malware and Vulnerabilility Assessment (DIMVA)*, 2009.

[33] Egele, M., Kruegel, C., Kirda, E., and Vigna, G., "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the 19th NDSS*, 2011.

[34] Enck, W., Gilbert, P., Chun, B.-G., Cox, L. P., Jung, J., McDaniel, P., and Sheth, A. N., "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX OSDI*, 2010.

[35] Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S., "A study of android application security," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[36] Enck, W., Ongtang, M., and McDaniel, P., "On lightweight mobile phone application certification," in *Proceedings of the 16th ACM CCS*, 2009.

[37] Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G., "Toward automated detection of logic vulnerabilities in web applications," in *Proceedings of the 19th USENIX Security Symposium*, 2010.

[38] Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D., "Android permissions demystified," in *Proceedings of the 18th ACM CCS*, 2011.

[39] Felt, A. P., Wang, H. J., Moshchuk, A., Hanna, S., and Chin, E., "Permission re-delegation: attacks and defenses," in *Proceedings of the 20th USENIX Security Symposium*, 2011.

[40] Ford, S., Cova, M., Kruegel, C., and Vigna, G., "Wepawet." http://wepawet.cs.ucsb.edu, 2009.

[41] Grace, M., Zhou, Y., Wang, Z., and Jiang, X., "Systematic detection of capability leaks in stock Android smartphones," in *Proceedings of the 19th NDSS*, 2012.

[42] Gu, G., Porras, P., Yegneswaran, V., Fong, M., and Lee, W., "BotHunter: Detecting malware infection through IDS-driven dialog correlation," in *Proceedings of 16th USENIX Security Symposium*, 2007.

[43] Gu, G., Zhang, J., and Lee., W., "Botsniffer: Detecting botnet command and control channels in network traffic.," in *Proceedings of the 15th Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[44] Gyöngyi, Z. and Garcia-Molina, H., "Web spam taxonomy," tech. rep., Stanford University, 2005.

[45] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H., "The weka data mining software: an update," *SIGKDD Explor. Newsl.*, vol. 11, no. 1, pp. 10–18, 2009.

[46] Hardy, N., "The confused deputy: (or why capabilities might have been invented)," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 4, pp. 36–38, 1988.

[47] Higgins, K. J., "'Aurora' exploit retooled to bypass Internet Explorer's DEP security." http://www.darkreading.com/security/-vulnerabilities/showArticle.jhtml?articleID=222301436.

[48] Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D., "These aren't the droids you're looking for: retrofitting android to protect data from imperious applications," in *Proceedings of the 18th ACM CCS*, 2011.

[49] Horwitz, S., Reps, T., and Binkley, D., "Interprocedural slicing using dependence graphs," *SIGPLAN Not.*, vol. 23, no. 7, pp. 35–46, 1988.

[50] Howard, F. and Komili, O., "Poisoned search results: How hackers have automated search engine poisoning attacks to distribute malware," tech. rep., SophosLab, 2010.

[51] Jain, S., Shafique, F., Djeric, V., and Goel, A., "Application-level isolation and recovery with solitude," in *Proceedings of ACM EuroSys*, 2008.

[52] John, J., Yu, F., Xie, Y., Abadi, M., and Krishnamurthy, A., "deSEO: Combating search-result poisoning," in *Usenix Security (to appear)*, 2011.

[53] Jovanovic, N., Kruegel, C., and Kirda, E., "Pixy: A static analysis tool for detecting web application vulnerabilities (short paper)," in *Proceedings of the IEEE S&P'06*, 2006.

[54] Liang, Z., Venkatakrishnan, V. N., and Sekar, R., "Isolated program execution: An application transparent approach for executing untrusted programs,"

[55] Lineberry, A., Richardson, D. L., and Wyatt, T., "These aren't permissions you're looking for," in *Proceedings of the Blackhat'10*, 2010.

[56] Livshits, V. B. and Lam, M. S., "Finding security vulnerabilities in java applications with static analysis," in *Proceedings of the 14th USENIX Security Symposium*, 2005.

[57] Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G., "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, (New York, NY, USA), pp. 229–240, ACM, 2012.

[58] Lu, L., Perdisci, R., and Lee, W., "Surf: detecting and measuring search poisoning," in *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, (New York, NY, USA), pp. 467–476, ACM, 2011.

[59] Lu, L., Yegneswaran, V., Porras, P., and Lee, W., "Blade: an attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, (New York, NY, USA), pp. 440–450, ACM, 2010.

[60] Martinez-Cabrera, A., "Malware infections double on web pages." http://articles.sfgate.com/2010-01-26/business/17836038_1_malware-infected-sites.

[61] Moshchuk, A., Bragin, T., Deville, D., Gribble, S. D., and Levy, H. M., "SpyProxy: Execution-based detection of malicious web content," in *Proceedings of 16th USENIX Security Symposium*, 2007.

[62] Moshchuk, A., Bragin, T., Gribble, S. D., and Levy, H. M., "A crawler-based study of spyware on the web," in *Network and Distributed System Security Symposium*, February 2006.

[63] Moshchuk, E., Bragin, T., Gribble, S. D., and Levy, H. M., "A crawler-based study of spyware on the web," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'06)*, 2006.

[64] Myers, A. C., "Jflow: practical mostly-static information flow control," in *Proceedings of the 26th ACM POPL*, 1999.

[65] Nazario, J., "phoneyc: A Virtual Client Honeyport," in *Proceedings of LEET*, 2009.

[66] Ntoulas, A. and Manasse, M., "Detecting spam web pages through content analysis," in *In Proceedings of the World Wide Web conference*, pp. 83–92, ACM Press, 2006.

[67] Oberheide, J., Cooke, E., and Jahanian, F., "Cloudav: N-version antivirus in the network cloud," in *Proceedings of 17th USENIX Security Symposium*, 2008.

[68] Page, L., Brin, S., Motwani, R., and Winograd, T., "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999.

[69] Provos, N., "Spybye - finding malware." http://www.monkey.org/p̃rovos/spybye/, 2009.

[70] Provos, N., Mavrommatis, P., Rajab, M. A., and Monrose, F., "All your iframes point to us," in *Proceedings of the 17th USENIX Security Symposium*, 2008.

[71] Provos, N., McNamee, D., Mavrommatis, P., Wang, K., and Modadugu, N., "The ghost in the browser analysis of web-based malware," in *1st Workshop on Hot Topics in Understanding Botnets*, 2007.

[72] Rajab, M. A., Ballard, L., Mavrommatis, P., Provos, N., and Zhao, X., "The nocebo effect on the web: an analysis of fake anti-virus distribution," in *Proceedings of the 3rd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more*, LEET'10, (Berkeley, CA, USA), pp. 3–3, USENIX Association, 2010.

[73] Ratanaworabhan, P., Livshits, B., and Zorn, B., "NOZZLE: A defense against heap-spraying code injection attacks," in *Proceedings of 18th USENIX Security Symposium*, 2009.

[74] Reis, C., Dunagan, J., Wang, H., Dubrovsky, O., and Esmeir, S., "Browsershield: Vulnerability driven filtering of dynamic html," in *Proceedings of OSDI*, 2006.

[75] Seifert, C., Steenson, R., Holtz, T., Yuan, B., and Davis, M. A., "Know your enemy: Malicious web servers." http://www.honeynet.org/papers/mws/, 2007.

[76] Seifert, C., Stokes, J., Lu, L., Heckerman, D., Colcernian, C., Parthasarathy, S., and Santhanam, N., "Scareware detection." US Patent Application 20120159620, 2011.

[77] Staiger, S., "Reverse engineering of graphical user interfaces using static analyses," in *Proceedings of the 14th IEEE WCRE*, 2007.

[78] Staiger, S., "Static analysis of programs with graphical user interface," in *Proceedings of the 11th IEEE CSMR*, 2007.

[79] Stiegler, M., Karp, A., Yee, K., Close, T., and Miller, M., "Polaris: virus-safe computing for Windows XP," *Communications of the ACM*, vol. 49, no. 9, p. 88, 2006.

[80] Thomas, K., Grier, C., Ma, J., Paxson, V., and Song, D., "Design and evaluation of a real-time url spam filtering service," in *In Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.

[81] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O., "TAJ: effective taint analysis of web applications," in *Proceedings of the ACM PLDI '09*, 2009.

[82] Urvoy, T., Chauveau, E., Filoche, P., and Lavergne, T., "Tracking web spam with html style similarities," *ACM Trans. Web*, vol. 2, pp. 3:1–3:28, March 2008.

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

[84] Wang, Y.-M., Beck, D., Jiang, X., and Roussev, R., "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Proceedings of the Network and Distributed Systems Security Symposium (NDSS'06)*, 2006.

[85] Wang, Y.-M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., and King, S., "Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities," in *Network and Distributed System Security Symposium (NDSS)*, 2006.

[86] Wassermann, G. and Su, Z., "Static detection of cross-site scripting vulnerabilities," in *Proceedings of the 30th ACM ICSE*, 2008.

[87] Wu, B. and Davison, B. D., "Identifying link farm spam pages," in *Proceedings of the 14th International World Wide Web Conference*, pp. 820–829, ACM Press, 2005.

[88] Wu, B. and Davison, B. D., "Detecting semantic cloaking on the web," in *Proceedings of the 15th international conference on World Wide Web*, WWW '06, (New York, NY, USA), pp. 819–828, ACM, 2006.

[89] Zeldovich, N., Boyd-Wickizer, S., Kohler, E., and Mazières, D., "Making information flow explicit in histar," in *Proceedings of the 7th USENIX OSDI*, 2006.

[90] Zhou, W., Zhou, Y., Jiang, X., and Ning, P., "DroidMOSS: Detecting repackaged smartphone applications in third-party android," in *Proceedings of ACM CODASPY'12*, 2012.

[91] Zhou, Y. and Jiang, X., "Dissecting android malware: Characterization and evolution," in *Proceedings of the IEEE Symposium on S&P'12*, 2012.

[92] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X., "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets," in *Proceedings of the 20th NDSS*, 2012.