UC Santa Barbara
UC Santa Barbara Electronic Theses and Dissertations

Title
Analyzing and Defending Against Evolving Web Threats

Permalink
https://escholarship.org/uc/item/4rf82369

Author
Kapravelos, Alexandros

Publication Date
2015

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA
Santa Barbara

Analyzing and Defending Against Evolving Web Threats

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Alexandros Kapravelos

Committee in Charge:

Giovanni Vigna, Chair

Christopher Kruegel

Richard Kemmerer

June 2015

The Dissertation of
Alexandros Kapravelos is approved:

Christopher Kruegel

Richard Kemmerer

Giovanni Vigna, Committee Chairperson

June 2015

Analyzing and Defending Against Evolving Web Threats

Copyright © 2015

by

Alexandros Kapravelos

Acknowledgements

I was fortunate enough to have some amazing people supporting me during this marathonian accomplishment that is called a PhD.

Chris and Giovanni shaped the researcher that I am today. They did not only teach me how to think with security in mind, but they also taught me a *research mentality* that is unique. They were always there for me, for every single ridiculous problem I had during those years, giving me the best advice that I could ever ask for. I learned so many things next to you that I will be always grateful for the influence you had in my life. I would like also to thank Dick for his incredible help, creative humor, and wisdom.

During my PhD I visited a lot of places that played a significant role in seeing how other groups work and how I can collaborate outside of seclab. I would like to thank Chris Grier, Vern Paxson, Stefan Savage, Geoff Voelker, Kurt Thomas and Elie Bursztein for mentoring me during my visits at ICSI, UC San Diego and Google. It was a fantastic experience and I had the unique opportunity to diversify my research and enrich it with the support of excellent researchers.

I found in seclab a second family. We worked together, we got rejected together, we resubmitted together. I could have never imagined the things that we achieved together when I joined this lab. In no particular order, I would like to thank: Adam, Gianluca, Yanick, Yan, Ali, Luca, Bob, Antonio, Jacopo, Dhilung, Fish, Kevin, Ludo, Manuel, Marco, Gregoire and

Nilo. I would like to thank also Clemens and Manu, for coping with me all these years. With friends like this I have nothing to fear.

I would never have been able to reach this point without the unconditional love and support from my Mom, Dad, and sister. You guys believed in me from the beginning and I will never forget that.

Curriculum Vitæ

Alexandros Kapravelos

Education

2015	Ph.D. Student in Computer Security Lab
	Computer Science Department
	University of California, Santa Barbara, USA
thesis	*Analyzing and Defending Against Evolving Web Threats*
supervisors	Professor Christopher Kruegel, Professor Giovanni Vigna
2010	M.Sc. Candidate in Distributed Computing Systems Lab
	Computer Science Department
	University of Crete, Greece
thesis	*Robust Prevention of Dial Attacks*
supervisor	Professor Evangelos Markatos
description	An extensive evaluation of the security properties that arise from making accessible telephone devices from the Internet through the use of VoIP. The term Dial stands for *Digitally Initiated Abuse of teLephones*
2007	B.Sc. in Computer Science
	Computer Science Department, University of Crete, Greece
thesis	*Packetloss: A Passive end-to-end Packet Loss estimation*

supervisor	Professor Evangelos Markatos
descriptio	A novel idea for estimating accurately the packet loss ratio between different measuring points.

Research Experience

2010 – 2015	Research Assistant, University of California, Santa Barbara Advisors: Giovanni Vigna, Christopher Kruegel
June – Sept 2014	Visiting PhD student, University of California, San Diego Advisors: Stefan Savage, Geoffrey Voelker
June – Sept 2013	Visiting PhD student, International Computer Science Institute at Berkeley Advisors: Vern Paxson, Chris Grier
2005 – 2010	Research Assistant, FORTH-ICS Advisor: Evangelos Markatos

Industry Experience

Sept – Dec 2014	Internship at Google
Oct – Dec 2013	Internship at Lastline Inc.

Abstract

Analyzing and Defending Against Evolving Web Threats

Alexandros Kapravelos

The browser has evolved from a simple program that displays static web pages into a continuously-changing platform that is shaping the Internet as we know it today. The fierce competition among browser vendors has led to the introduction of a plethora of features in the past few years. At the same time, it remains the *de facto* way to access the Internet for billions of users. Because of such rapid evolution and wide popularity, the browser has attracted attackers, who pose new threats to unsuspecting Internet surfers.

In this dissertation, I present my work on securing the browser against current and emerging threats. First, I discuss my work on honeyclients, which are tools that identify malicious pages that compromise the browser, and how one can evade such systems. Then, I describe a new system that I built, called *Revolver*, that automatically tracks the evolution of JavaScript and is capable of identifying evasive web-based malware by finding similarities in JavaScript samples with different classifications. Finally, I present Hulk, a system that automatically analyzes and classifies browser extensions.

Contents

Acknowledgements iv

Curriculum Vitæ vi

Abstract viii

List of Figures xii

List of Tables xiv

1 Introduction 1
- 1.1 Drive-by Downloads . . . 3
- 1.2 Browser Extensions . . . 8
- 1.3 Contributions . . . 10

2 Related Work 12
- 2.1 Evading High-interaction Honeyclients . . . 12
- 2.2 Detecting Evasive Web-based Malware . . . 16
- 2.3 Analyzing Browser Extensions . . . 20

3 Evading High-Interaction Honeyclients 22
- 3.1 Honeyclients . . . 24
  - 3.1.1 Security requirements for high-interaction honeyclients . . . 25
  - 3.1.2 Design choices for high-interaction honeyclients . . . 27
  - 3.1.3 Honeyclients in practice . . . 29
- 3.2 Attacks Against Honeyclients . . . 31
  - 3.2.1 Fingerprinting the environment . . . 32
  - 3.2.2 Evading detection . . . 37

3.2.3 Summary . . . 44
3.3 Attacks in the Real World . . . 44
3.4 Countermeasures . . . 48
3.4.1 Transparency . . . 49
3.4.2 Protection of the monitoring system . . . 50
3.5 Conclusion . . . 51

4 An Automated Approach to the Detection of Evasive Web-based Malware 54
4.1 Background and Overview . . . 59
4.2 Approach . . . 65
4.2.1 Oracle . . . 67
4.2.2 Abstract Syntax Trees . . . 69
4.2.3 Similarity Detection . . . 70
4.2.4 Optimizations . . . 74
4.2.5 Classification . . . 76
4.3 Implementation . . . 80
4.4 Evaluation . . . 83
4.4.1 Evasions in the wild . . . 83
4.4.2 Evasions case studies . . . 86
4.5 Discussion . . . 93
4.6 Conclusions . . . 95

5 Eliciting Malicious Behavior in Browser Extensions 97
5.1 Background . . . 100
5.1.1 Chrome Extension Composition . . . 100
5.1.2 Installing Extensions . . . 101
5.1.3 Extension Permissions . . . 102
5.2 Architecture . . . 105
5.2.1 Profiling Extensions . . . 106
5.2.2 Event-Based Execution . . . 108
5.2.3 Detecting Malicious Behavior . . . 110
5.2.4 Injected Content Analysis . . . 114
5.3 Results . . . 115
5.3.1 Permissions Used . . . 118
5.3.2 Network Level . . . 120
5.3.3 Extensions Management . . . 121
5.3.4 Code Injection . . . 121
5.4 Profiting from Maliciousness . . . 122
5.4.1 Ad Manipulation . . . 123
5.4.2 Affiliate Fraud . . . 125

5.4.3 Information Theft 129
5.4.4 OSN Abuse 130
5.5 Recommendations 132
5.6 Limitations 133
5.7 Conclusions 135

6 Conclusions and Future Work **140**

Bibliography **142**

List of Figures

1.1 Control panel of an exploit kit. 6

3.1 Malicious code interaction with the Honeyclient system. 25
3.2 Capture-HPC Architecture 29
3.3 Detecting Capture-HPC presence in the file system with JavaScript. . . 34
3.4 Function hooks detection: before calling a critical function, we check if it is hooked. 36
3.5 Browser cache poisoning attack. 40
3.6 In-memory keylogger: collects keystrokes and sends them to the attacker with HTTP GET requests. 42
3.7 Confuse honeyclient: find an Internet Explorer instance and force it to visit a URL of our choice. 53

4.1 Malicious code that sets up a shellcode. 61
4.2 An evasion using non-existent ActiveX controls. 62
4.3 Architecture of *Revolver*. 66
4.4 Data structures used by *Revolver*. 70
4.5 Number of detected similarities as a function of the distance threshold. . 82
4.6 The resulting amount of similarities for different similarity thresholds. . 82
4.7 Evasion based on differences in the scope handling inside `eval` in different JavaScript engines. 87
4.8 Timeline of PDF evasions automatically detected by *Revolver*. 88
4.9 Evasion based on the ability to access the `eval` function as a property of native objects in Adobe's JavaScript engine. 89
4.10 Evasion based on PDF specific objects *app* and *target*. 89
4.11 An evasion taking advantage of a subtle bug in Wepawet's JavaScript engine in order to protect the XOR key. 96

5.1 Example of a manifest that shows API permissions for two hosts, followed by content scripts that run on `http://www.yahoo.com`, followed by a background script that runs on all pages. Finally, the CSP specifies the ability to include and `eval` scripts in the extension from `foo.com`. 103
5.2 Permission-related JSON from the manifest file of an extension performing ad replacement. 124
5.3 Permissions and content script excerpts from the manifest for an extension that spams on Facebook and creates Tumblr accounts. 131

List of Tables

3.1 Summary of the attacks: a ✗ indicates that the attack did not evade the honeyclient, a ✓ indicates that the attack was not detected. 45
3.2 Capture-HPC and Wepawet analysis results. 45
3.3 Possible JavaScript evasion techniques against Capture-HPC found in the wild. 46

4.1 Candidate pairs classification (B is a benign sequence, M is a malicious sequence, $*$ indicates a wildcard value). 77
4.2 Benign scripts from Wepawet that have similarities with malicious scripts and their classification from *Revolver*. 84

5.1 Classification distribution of extensions. 115
5.2 Distribution of detected suspicious/malicious behavior from analyzed extensions. Notice that an extension might have more than one detections and that we mark with [m] detections classified as malicious and with [s] detections classified as suspicious. 116
5.3 The top 10 permissions found in the manifest files for all extensions we ran. Extensions can include more than one permission. 117
5.4 The top 25 host permissions used by extensions. Extensions can include more than one host permission per manifest. 137
5.5 The top 25 hosts used in extensions' content script permissions. 138
5.6 The top 15 chrome.* APIs called by extensions during dynamic analysis. 139

Chapter 1

Introduction

Today the Internet plays an essential part in the lives of billions of people, who use the Internet to find information, to communicate, or for entertainment. The browser has become our portal to the Internet and it is so significant to our daily operations that it is coming pre-installed in every modern operating system. Over the past two decades, the browser has evolved significantly and it is the main element that drives innovation on the web. It even affects our lives outside of the digital world, as many corporations use the browser as their main platform to do business online, changing traditional business models that have lasted for centuries.

With such an important role it was inevitable that miscreants would quickly get interested in attacking the browser. There are three main reasons that make the browser attractive for attackers: evolution, input, and connectivity. The browser is a unique piece of software that evolves constantly. For example, Google's browser Chrome has a new version released every 6 weeks [36]. The input to the browser comes from its user in the form of a URL,

which indicates the page that the user wants to visit. But to display that page, the browser fetches third-party code that gets rendered locally, allowing this way the site developer to manipulate the browser. More importantly, this happens over a network, which means that the browser is not isolated on a disconnected machine, rather it is accessible via a network. This gives attackers a great platform to abuse in many different ways, especially because it is possible to profit from exploiting the browser and collecting valuable personal information.

In this dissertation I present my work on understanding in depth attacks that target the browser and on building defenses to protect the users. I focus on two specific types of attacks: drive-by downloads and malicious browser extensions. Drive-by downloads are attacks that exploit a vulnerability in the browser or its plugins and gain full control of the system by hijacking the browser's execution flow. It is a way for the attacker to introduce new binary code into the user's system and compromise its security. Browser extensions are small JavaScript/HTML programs that reside inside the browser and are limited in power. Nevertheless, since browser extensions are capable of manipulating the browser and the pages that the user visits, they are an effective tool to steal personal information and perform other attacks.

1.1 Drive-by Downloads

Drive-by downloads have troubled both the academic community and the browser vendors for more than a decade. They have been very successful, because the user does not need to interact with them in any way: the user just needs to visit a page containing malicious code to get infected. Because most people browse the web by clicking on any link that draws their interest, it is not easy to avoid such pages, and, eventually, the browser is silently attacked.

There are three ways to deliver a drive-by download to an unsuspecting user. A drive-by download can be hosted on a web server controlled by the attacker. To lure users to their malicious sites, the attackers can promote their malicious links in many ways. For example, links to malicious pages can be distributed leveraging social networks, by posting links in the trending topics. Another way to attract users to malicious pages is by sending emails that entice the user to click on the link. Malicious pages can also be returned in search results because the attacker creates carefully crafted (and optimized) web pages that appear to be related to very common, trending terms (this type of activity is also known as Black Hat Search Engine Optimization, or Black Hat SEO).

Another way to deliver drive-by downloads is to compromise an already existing website and infect the page's source code. This requires the attacker to compromise the security of the website by finding a vulnerability that allows the attacker to inject new or modify existing web content. Once the site has been compromised, any user who visits the previously-benign

website can be targeted by a drive-by download. While cleaning up a compromised site is not difficult, finding and removing the underlying vulnerability can be challenging. If the vulnerability is not removed, it allows the attacker to re-infect the vulnerable website. Another benefit that comes to the attacker when he/she compromises an existing website is that the domain that is now hosting malicious code has a previously good reputation, and, therefore, it is less likely that the site will get blacklisted by software that protects against drive-by downloads using reputation metadata. In addition, the attacker does not need to attract users to the site, as the site has its own regular visitors.

Finally, one can serve drive-by downloads by taking advantage of the openness of the web. A website has several external dependencies, such as script and iframe inclusions. These are components of the site that load third-party code to enhance the browsing experience or to monetize through advertisements. An attacker can either compromise the site that is hosting the third-party code or register a malicious advertisement that ends up being served from a benign website. These attacks are interesting because they do not rely on the security of the website that the user is visiting but rather on the security of the dependencies that the website has.

Anatomy of Drive-by Downloads. An attack against the browser starts once the unsuspecting user visits a site containing malicious code. At that point, the specially crafted page targets several vulnerabilities against the browser and its plugins. The first step of the attack is to determine if the current browser configuration is vulnerable. If the software is

found to be vulnerable, then the malicious code will try to exploit the detected vulnerability. These are often memory corruption bugs, such as buffer overflows or use-after-free flaws. The attacker is able to hijack the execution flow of the browser by first introducing additional executable code in the memory, for example through a JavaScript string variable, and then using a vulnerability to direct the execution there. The new code segment that is injected in the browser by the attacker is called *shellcode*, and, in drive-by downloads, it is used as the first stage of the attack. Its role is to infect the victim's machine with malware, persistently compromising the system. The types of malware and what happens after the system is compromised is out of the scope of this dissertation, as we focus on *how a browser can be compromised*.

Exploit Kits. With such diverse population of Internet users, a malicious page needs to support multiple client configurations in order to compromise a significant amount of victims. For example, as of March 2015, there are still 1% of Internet users that have Internet Explorer 6 [1], a Microsoft browser that was released in 2001 and was deprecated in 2011 [2]. This diversity of vulnerable targets leads to an interesting evolution of drive-by downloads with the rise of tools that offer multiple attacks as a service, namely *exploit kits*. These tools abstract the hard work of incorporating multiple exploits for several browsers or its plugins and also hosting the malicious code on web servers. They come with a control panel, as shown in Figure 1.1, where the customer can see statistics about its victims and configure various parameters of the exploit kit. This configuration involves distributing

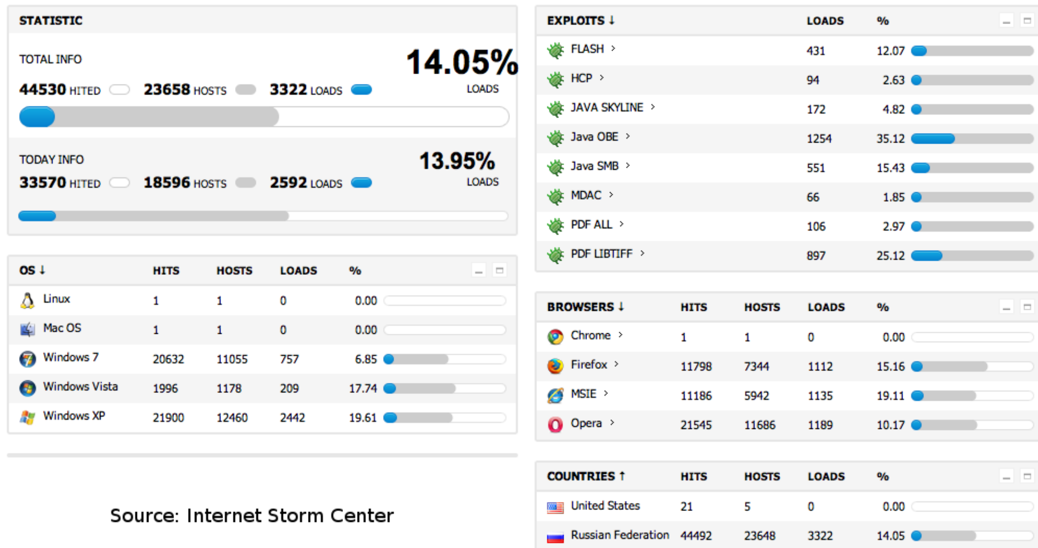

Figure 1.1: Control panel of an exploit kit.

customized malware, often checked by the exploit kit to verify that it does not get detected by any antivirus software. This evolution of providing drive-by downloads as a service resulted in the commercialization of such attacks, where a novice attacker can deploy sophisticated browser attacks without any knowledge on how to write an exploit targeting a browser. Malwarebytes reports that two thirds of new malware they observe originates from exploit kits [63], which underlines the importance of understanding such attacks and finding effective ways to mitigate them.

Defenses. Analyzing a web page to determine if it contains malicious code that will perform a drive-by download attack is not a trivial task. The attackers are deliberately crafting the page in such a way so that their attack is not easily detectable. The malicious code is often not directly located in the original page, but it is obfuscated and dynamically generated

as the browser renders the page. This means that by observing the web page statically or as it appears on the network, it is very hard to determine if it is malicious or not. In order to have the full picture and all components of the page, an analysis system has to dynamically render the page, so that all generated code is revealed. To do this, researchers have designed and developed *honeyclients*, specially-crafted browsers that provide detailed information about the actions performed by the code in a web page when it is rendered in the browser, and the honeyclients are able to determine if the web page is malicious. Honeyclients are split into two basic categories: high-interaction and low-interaction honeyclients.

High-interaction honeyclients resemble as much as they can a real client visiting web pages. They use a real browser running on an operating system exactly as a user would, but the system is enhanced so that it can detect any changes that are happening. By inspecting the state of the system, high-interaction honeyclients can infer an infection by observing any changes to the system, like unknown processes that are spawned after visiting a web page, or by changes in the registry and the filesystem [70, 71, 81, 99, 106].

Part of this dissertation focuses on understanding the limitations of honeyclients. As discussed in Chapter 3, there are several ways in which an attacker might be able to evade a high-interaction honeyclient so that his/her attacks remain unnoticed. For example, the attacker can choose to perform an in-memory attack, sacrificing persistence on the victim's system in favor of stealthiness. In fact, the Angler exploit kit has already demonstrated such capabilities in 2014 [64].

Low-interaction honeyclients are tools that provide deeper insight into what is happening when a browser visits a web page. This can be achieved by emulating part or the whole functionality of a browser, which gives significant flexibility into exporting enough information from a page visit so that the system can accurately identify a drive-by download. For example, an emulated browser could export all JavaScript variables, so that they can be analyzed to detect the presence of binary code, something that should be encountered only in a drive-by download. One significant drawback of low-interaction honeyclients is the fact that they might diverge too much from a real browser. In fact, through our work we found samples in the wild that take advantage specifically of the fact that their attack code is being analyzed. In such cases, the samples were not identified correctly as malicious, as the attack was never launched after detecting the presence of an analysis system. To cope with this problem, we have developed a system called *Revolver* that is able to track the evolution of JavaScript found in web pages. As further discussed in Chapter 5, by using the classification results from an oracle and the similarity between scripts, we are able to identify automatically evasive drive-by downloads.

1.2 Browser Extensions

We spend increasingly more time using browsers in our daily lives to perform sensitive tasks such as banking, authoring documents, accessing our medical records, and interacting

socially. Modern browsers support modification through extensions to make a user's interaction with the web easier, customizable, and to enable richer content. Browser extensions are small JavaScript/HTML programs that reside inside the browser and have access to a privileged browser API compared to websites. Through an extension the user can change not only how a website appears, but also enhance the browsing experience with additional client-side functionality. For example, one of the most popular extensions is AdBlock, an extension that has a blacklist of domains and URL patterns of advertising companies, and when browsing the web it removes the components of the visited pages that match the blacklist, resulting in an ad-free browsing experience. The extensions do not have unrestricted access to every page that the user is visiting, but in every browser there is a permission model that they need to comply with. Similar to what happens with mobile application stores, browser vendors maintain and control the available extensions through extension stores, like Chrome's Web Store.

Recently, we have encountered a shift in the attackers' interests towards extensions, which seems to be for practical and economical reasons: extensions are easy to write, as they require less skill than writing a drive-by download. At the same time, they are very profitable, as they can control the browser and the pages that the user is visiting. Interestingly, we have seen users infected with malware where the malicious binary was side-loading a malicious extension in the browser, because, by doing this, it is easier to manipulate the user's browsing experience. We have also found extensions installed by millions of users that performed

malicious activities resulting in millions of US dollars in revenue for the extension authors. Through the use of browser extensions, the attackers are able to perform malicious activities without an exploit and gain access to powerful APIs and data available through the browser extension platform. Malicious extensions can intercept HTTP requests, as well as modify the functionality and appearance of all web content, which makes them very dangerous when used by miscreants. To understand and mitigate this new threat against the browser, we built a system called *Hulk*, presented in Chapter 5, that uses dynamic analysis techniques to analyze browser extensions and identify those containing malicious functionality.

1.3 Contributions

In this dissertation, I make the following contributions to securing the browser:

- I study how high-interaction honeyclients work and identify major design flaws in how drive-by downloads are currently detected. Several attacks that evade detection in four popular high-interaction honeyclients were implemented.
- I built a system, called *Revolver*, that tracks the evolution of JavaScript. By leveraging clustering and program analysis techniques applied to millions of JavaScript files, we are able to automatically pinpoint previously-unknown evasion attacks.
- I built a system, called *Hulk*, that analyzes browser extensions and identifies malicious ones. Our system is able to elicit malicious behavior from the browser extension by using

novel techniques, such as *HoneyPages* and *event handler fuzzing*. Our work resulted in removing extensions from Google's Web Store that were affecting millions of users.

Chapter 2

Related Work

Drive-by downloads and the security implications of browser extensions have been extensively studied in the literature. In this chapter, we will focus specifically on systems that analyze attacks against the browser and how they can be evaded, how such evasive web-based malware can be detected and what research has been done regarding browser extensions.

2.1 Evading High-interaction Honeyclients

Our work is mainly related to the problems of identifying weaknesses in the defensive systems designed to monitor and detect the execution of malicious programs, and of devising attacks against them. Here, we will review the current state-of-the-art in these areas, focusing in particular on systems that detect web-based and binary malware and on intrusion detection tools.

Web-based malware monitors. Attacks against high-interaction honeyclients have been previously discussed. In particular, Wang et al. discuss three avenues to evade their HoneyMonkey system [106]: *(i)* identifying HoneyMonkey machines (based on detecting their IP addresses, by testing whether the browser is driven by a human, or by identifying the presence of a virtual machine or the HoneyMonkey code itself); *(ii)* running exploits that do not trigger HoneyMonkey's detection (e.g., by using time delays); and *(iii)* randomizing the attack (trading off infection rates for detection rates).

We build on and extend this research in several ways. First, we have implemented the aforementioned attacks and confirmed that they are (still) effective against all current, publicly-available honeyclient systems. Second, we introduce and discuss in detail novel attacks against high-interaction honeyclients, with the goal of providing simple and practical implementations. Finally, we discuss the design trade-offs of these attacks. For example, we show how to detect the presence of a honeyclient from a page's JavaScript and from an exploit's shellcode. JavaScript-based attacks have more limited capability because they are restricted by the JavaScript security model (e.g., they cannot be used to detect hooks in the memory of a process), but they are more difficult to detect by current honeyclients, because they do not cause any changes on the attacked system (e.g., no new file is created and no exploit is launched).

We also note that Wang's paper concludes its discussion of possible countermeasures by introducing the Vulnerability-Specific Exploit Detector (VSED), a tool that checks the

browser with vulnerability-specific predicates to determine when an attack is about to trigger a vulnerability. VSED presents a significant deviation from the traditional state-change-based approach for detecting drive-by-download attacks. In fact, while state-change-based approaches focus on detecting the consequences of a successful drive-by-download, VSED attempts to detect the actual exploitation.

Some of the attacks identified in [106] have become standard in several drive-by-download toolkits. In particular, it is common for these kits to launch an attack only once per visiting client IP [82], to only attack clients coming from specific geographic regions (determined on the basis of GeoIP location data) [13], and to avoid attacking IPs known to belong to security researchers and security companies [42]. Another attack against detection tools used by some drive-by-download campaigns consists of waiting for some minimal user interaction before launching the exploit. For example, the JavaScript code used by Mebroot triggers only when the user clicks on a page's link or releases the mouse anywhere on the page.

Malware sandboxes. Binary malware is a significant security threat, and, consequently, a large body of work exists to analyze and detect malicious code. Currently, the most popular approach for malware analysis relies on dynamic analysis systems, often called sandboxing [6, 11, 21, 46, 78, 100]. A sandbox is an instrumented execution environment that runs a potentially malicious program, while monitoring its interactions with the operating system and other hosts. Similar to honeyclients, malware sandboxes execute unknown code and determine its maliciousness based on the analysis of its behavior.

Since system emulators and virtual machines are commonly employed to implement sandboxes, malware authors have developed a number of techniques to identify them (and, in turn, avoid the detection of the monitoring system). For example, a number of instructions have been identified that behave differently on a virtualized or emulated environment than on a real machine [29, 65, 79, 85, 94]. This has led researchers to design monitoring systems that are transparent to malware checks (i.e., that cannot be easily distinguished from regular hosts), by either removing artifacts of regular monitoring tools [59] or by introducing mechanisms (such as virtualization and dynamic translation) that by design remain transparent to a wider range of checks [23, 103].

Another class of attacks against a malware sandbox consists of detecting, disabling, or otherwise subverting its monitoring facilities. These threats have prompted researchers to experiment with new designs for monitoring systems, in which the monitoring components are protected by isolating them from the untrusted monitored environment through hardware memory protection and virtualization features ("in-VM" designs) [97] or by removing them from the monitored environment ("out-of-VM" designs) [45].

In this dissertation we dissect the monitoring and isolation mechanisms employed in high-interaction honeyclients. As we will see in Chapter 3, many of the approaches currently used are vulnerable to attacks similar to those devised against malware monitoring systems.

Intrusion detection systems. Our work continues the line of research on attacking tools designed to detect malicious activity, in particular, intrusion detection systems (IDSs). A

few notable results include Ptacek and Newsham's attacks against network IDSs [83], Fogla and Lee's evasion attacks against anomaly-based IDSs [31], Vigna et al.'s approach to evade signature-based IDSs [104], and Van Gundy et al.'s "omission" attack against signature generation tools for polymorphic worms [102]. Our research identifies high-interaction honeyclients as a new, important target for offensive techniques, and shows weaknesses in several popular implementations.

2.2 Detecting Evasive Web-based Malware

Detection of evasive code. The detection of code that behaves differently when run in an analysis environment than when executed on a regular machine is a well-known problem in the binary malware community. A number of techniques have been developed to check if a binary is running inside an emulator or a virtual machine [29, 85, 94]. In this context, evasive code consists of instructions that produce different results or side-effects on an emulator and on a real host [66, 79]. The original malware code is modified to run these checks: if the check identifies an analysis system, the code behaves in a benign way, thus evading detection.

Researchers have dealt with such evasive checks in two ways. First, they have designed systems that remain transparent to a wide range of malware checks [24, 103]. Second, they have developed techniques to detect the presence of such checks, for example by comparing

the behavior of a sample on a reference machine with that obtained by running it on an analysis host [8, 47, 58].

Similar to the case of evasions against binary analysis environments, the results produced by honeyclients (i.e., the classification of a web page as either malicious or benign) can be confused by sufficiently-sophisticated evasion techniques. Honeyclients are not perfect and attackers have found ways to evade them [48, 86, 106]. For example, malicious web pages may be designed to launch an exploit only after they have verified that the current visitor is a regular user, rather than an automated detection tool. A web page may check that the visitor performs some activity, such as moving the mouse or clicking on links, or that the browser possesses the idiosyncratic properties of commonly-used modern browsers, rather than being a simple emulator. If any of these checks are not satisfied, the malicious web page will refrain from launching the attack, and, as a consequence, will be incorrectly classified as benign, thus evading detection.

The problem of evasive code in web attacks has only recently been investigated. Kolbitsch et al. [52] have studied the "fragility" of malicious code, i.e., its dependence for correct execution on the presence of a particular execution environment (e.g., specific browser and plugin versions). They report several techniques used by malicious code for environment matching: some of these techniques may well be used to distinguish analysis tools from regular browsers and evade detection. They propose ROZZLE, a system that explores multiple execution paths in a program, thus bypassing environment checks. Rozzle only detects

fingerprinting that leverages control flow branches and depends upon the environment. It can be evaded by techniques that do not need control-flow branches, e.g., those based on browser or JavaScript quirks. For example, the property `window.innerWidth` contains the width of the browser window viewport in Firefox and Chrome, and is undefined in Internet Explorer. Therefore, a malicious code that initialized a decoding key as *xorkey=window.innerWidth*0+3* would compute a different result for xorkey in Firefox/Chrome (3) and IE (Not a Number error), and could be used to decode malicious code in specific browsers. Rozzle will not trigger its multi-path techniques in such cases and can be evaded.

Revolver takes a different approach to identifying evasive code in JavaScript programs. Instead of forcing an evasive program to display its full behavior (by executing it in parallel on a reference host and in an analysis environment [8], or by forcing the execution through multiple, interesting paths [52]), it leverages the existence of two distinct but similar pieces of code and the fact that, despite their similarity, they are classified differently by detection tools. In addition, *Revolver* can precisely and automatically identify the code responsible for an evasion.

JavaScript code analysis. In the last few years, there have been a number of approaches to analyzing JavaScript code. For example, Prophiler [14] and ZOZZLE [19] have used characteristics of JavaScript code to predict if a script is malicious or benign. ZOZZLE, in

particular, leverages features associated with AST context information (such as, the presence of a variable named `scode` in the context of a loop), for its classification.

Cujo [91] uses static and dynamic code features to identify malicious JavaScript programs. More precisely, it processes the static program and traces of its execution into q-grams that are classified using machine learning techniques.

Revolver performs the core of its analysis statically, by computing the similarity between pairs of ASTs. However, *Revolver* also relies on dynamic analysis, in particular to obtain access to the code generated dynamically by a script (e.g., via the `eval()` function), which is a common technique used by obfuscated and malicious code.

Code similarity. The task of automatically detecting "clones," i.e., segments of code that are similar (according to some notion of similarity), is an established line of work in the software engineering community [80, 93]. Unfortunately, many of the techniques developed here assume that the code under analysis is well-behaved or at least not adversarial, that is, not actively trying to elude the classification. Of course, this assumption does not hold when examining malicious code.

Similarity between malicious binaries has been used to quickly identify different variants of the same malware family. The main challenge in this context is dealing with extremely large numbers of samples without source code and large feature spaces from runtime data. Different techniques have been proposed to overcome these issues: for example, Bayer et

al. [10] rely on locality sensitive hashing to reduce the number of items to compare, while Jong et al. [44] use feature hashing to reduce the number of features.

As a comparison, *Revolver* aims not only to identify pieces of JavaScript code that are similar, but also to understand why they differ and especially if these differences are responsible for changing the classification of the sample.

2.3 Analyzing Browser Extensions

Browser extensions have been available for Internet Explorer and Firefox for over a decade. As a result of a study of vulnerabilities in Firefox extensions, Barth et al. designed an extension architecture that promotes least privilege and isolation of components to prevent a compromised extension from gaining full access to a user's browser [9], an architecture subsequently adopted by Google Chrome. Since then, further work has examined the success of the Chrome extension architecture at preventing damage [15] and the ability of developers to correctly request privileges for their extensions [28]. Similar studies have examined the Firefox extension system to limit the potential damage arising from exploitation of extension vulnerabilities, and to improve the defenses the browser provides [98]. These works have a focus mostly tangential to our work, since the principle of least privilege does not prevent an overtly malicious extension from executing malicious code.

The security industry has documented malicious extensions in ways similar to malware reports and other new threats [4, 7]. Liu et al. examined Google Chrome extensions and, based on malicious extensions the authors built, suggested refined privileges to make detecting malicious extensions easier [60]. In our work, we build a system that performs dynamic analysis and classification of extensions and present an analysis of malicious extensions that we found in the wild.

JavaScript-based program analysis has particular promise for benefiting our work, and in light of our current limitations we will be exploring techniques that we can adapt to improve our system's detection capabilities. Research has applied information flow analysis to Firefox extensions [22], performed taint-based tracking of untrusted data within the browser [25], used symbolic execution to detect vulnerabilities [95], applied static verification to extensions [40], contained extensions in privacy-preserving environments [57], and used supervised learning of browser memory profiles to detect privacy-sensitive events [35].

Our work has similarities to that of other malware detection and execution systems. While our implementation and requirements significantly differ from systems that execute Windows binary malware (such as Anubis [6]), at a high level we share common goals of executing and extracting data from samples. Like Anubis, Wepawet, the GQ honeyfarm, and other malware execution platforms, we share the difficult problem of triggering malicious behavior in a synthetic environment [18, 55]. Other research in this area have focused on classification and discerning malware from goodware [87].

Chapter 3

Evading High-Interaction Honeyclients

First,we examine the security model that high-interaction honeyclients use and evaluate their weaknesses in practice. By analyzing in depth how these analysis systems work we show in this chapter multiple ways that a miscreant can use to craft evasive attacks and avoid detection.

Drive-by-download attacks are one of the most pervasive threats on the web, and past measurements have found millions of malicious web pages [12, 81]. In addition, studies have shown that a large portion of the online population uses software that is vulnerable to the exploits used in drive-by-download attacks [32].

A primary line of defense against drive-by-download attacks consists of detecting web pages that perform such attacks and publishing their addresses on blacklists. Then, browsers can consult these blacklists and block requests to pages that are known to be malicious. This mechanism is currently used in all major browsers, typically by querying Google's Safe Browsing API or Microsoft's SmartScreen Filter [37, 67].

The approach used in high-interaction honeyclients focuses on detecting the side-effects of a successful exploit (i.e., the changes to the underlying system), rather than detecting the exploit itself, an approach that some refer to as detection" [106]. While this approach has merits (e.g., it provides very convincing evidence of the maliciousness of a detected page), it also creates an opportunity to attack the detection system. More precisely, an attacker can use the window between the launching of an exploit and the execution of its actual drive-by component (whose effects are detected by a high-interaction honeyclients) to attack and evade the honeyclient.

In this chapter, the security model of high-interaction honeyclients is put under the microscope and its weaknesses are evaluated in practice. More precisely, we first review high-interaction honeyclients in general, discussing different possible designs and their security properties. We then introduce a number of possible attacks that leverage weaknesses in the design of high-interaction honeyclients to evade their detection. Finally, we implement these attacks and test them against four popular, well-known implementations of high-interaction honeyclients. Our attacks allow malicious web pages to avoid being detected by a high-interaction honeyclient, while continuing to be effective against regular visitors. Some of these attacks have been previously described; nevertheless, we show concrete implementations that successfully bypass well-known, commonly-used honeyclient tools. In addition, we introduce three novel honeyclient attacks (JavaScript-based honeyclient detection, in-memory execution, whitelist-based attacks) that enable us to detect the presence

of a high-interaction honeyclient or to perform a drive-by-download without triggering the honeyclient's detection mechanisms.

We also note that it is relatively easy to retrofit existing drive-by-download toolkits with the evasion techniques that we present here. This makes their impact even more worrisome, and it increases the urgency for implementing adequate defensive mechanisms in high-interaction honeyclients.

3.1 Honeyclients

High-interaction honeyclients use a full-featured web browser to visit potentially malicious web pages. The environment in which the browser runs is monitored to determine if the visit resulted in the system being compromised. In particular, the honeyclient records all the modifications that occur during the visit of a page, such as files created or deleted, registry keys modified, and processes launched. If any unexpected modification occurs, this is considered as the manifestation of an attack, and the corresponding page is flagged as malicious.

In this section, we first describe the security requirements for honeyclients. Then, we discuss the key design choices in the development of honeyclients, and we conclude examining in detail a specific honeyclient implementation.

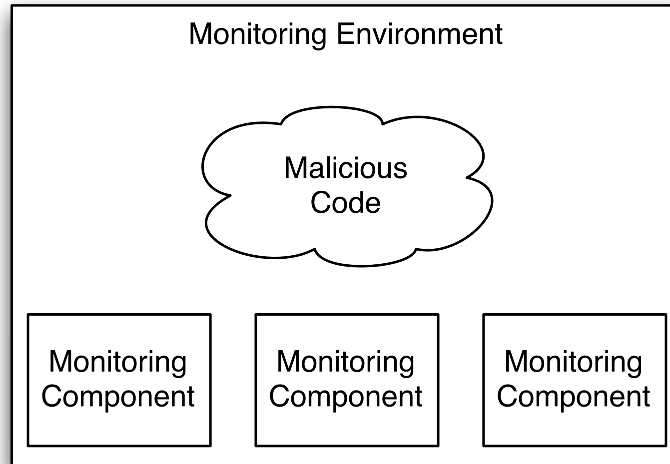

Figure 3.1: Malicious code interaction with the Honeyclient system.

3.1.1 Security requirements for high-interaction honeyclients

An ideal honeyclient system would be capable of detecting all the malicious web pages that it visits. There are three general reasons that may cause a missed detection: 1) the honeyclient is not exploitable, thus the attack performed by the malicious web page is not successful; 2) the honeyclient is incapable of monitoring the changes caused by a successful attack, thus the attack is not detected; and 3) the presence of the honeyclient is detected by the malicious pages, thus the attack is not run.

The first issue (the honeyclient must be vulnerable) can be addressed through careful configuration of the honeyclient system. Old, vulnerable versions of browsers and operating systems are used, and a large number of additional components (plugins and ActiveX)

are installed on the system, to maximize the possibility of successful exploits[1]. Even if this configuration is a complex task, in the rest of this chapter, we will assume that the honeyclient system is vulnerable to at least one of the exploits launched by a malicious page.

Second, effective monitoring requires that the monitoring facilities used by the honeyclient system cannot be bypassed by an attack. The well-known reference monitor concept [43] describes a set of requirements that security mechanisms must enforce to prevent tampering from the attacker and to ensure valid detection of the malicious activity:

Provide complete mediation: The monitoring mechanism must always be invoked, when a potentially malicious URL is tested on the system. It is essential in the case of honeyclients that the mechanism is able to detect all the possible changes that a successful attack could produce on the targeted system.

Be tamperproof: The monitoring mechanism should not be susceptible to tampering. For the honeyclient, this means that the malicious code should not be able to affect it in any way. For example, if the malicious code were able to kill the monitoring process or to blame another URL for the malicious activity, the reference monitor would be useless.

Be verifiable: The monitoring mechanism should be easy to verify for completeness and correctness. Unfortunately, this might not be an easy task, given the complexity of today's

[1] In [18] we showed that this approach has some inherent limitations, as there is a large number of vulnerable plugins, some of which may be incompatible with each other. Therefore, it may be impractical to create an environment that is vulnerable to all known attacks.

honeyclients, which include large operating systems (e.g., Windows) and applications (browsers).

A third venue of evasion is related to the **transparency** [33] of a high-interaction honeyclient. The honeyclient system should be indistinguishable from a regular host, to prevent malicious web pages from behaving differently inside a monitoring environment than on a real host.

3.1.2 Design choices for high-interaction honeyclients

Given the requirements described above, there are a few important design choices that can be made when developing a high-interaction honeyclient.

A first design choice is the placement of the monitoring mechanism inside or outside the guest environment executing the browser process. This “in-VM” vs. “out-of-VM” choice is a well-known and widely-discussed aspect of any malware analysis environment. Developing the monitoring mechanisms within the guest operating system greatly simplifies the architecture of the system, but, at the same time, makes the system vulnerable to detection, as the artifacts that implement the monitoring infrastructure cohabitate with the malicious code. By implementing the monitor at the kernel-level it is possible to better control access to the monitoring artifacts (drivers, processes, etc.) However, this is at the cost of increased complexity. In addition, there exist honeynet vulnerable configurations in which the code

that attacks the browser is able to gain access to kernel-level data structures. In this case it might be hard to hide the presence of the monitoring artifact from the malicious code.

We believe that a more appropriate model for honeyclients requires that the monitoring system is completely isolated from the environment. By moving the inspection of the potentially malicious code outside the virtual machine we guarantee that the attacker cannot tamper with the system. In practice, this is not trivial to implement and there are several obstacles to overcome, in order to have a deep insight of the program's execution inside the guest OS without compromising speed. We discuss in more detail the practical implications of running the monitoring system inside a virtual machine in Section 3.4, and we propose several methods on how to overcome the limitations of this approach.

Another design choice is the type and granularity of monitoring. This is a challenge especially in Windows-based system, because the Windows OS has a very large number of mechanisms for interacting with processes, injecting code, modifying files, etc. and therefore it is not easy to create a monitoring infrastructure that is able to collect the right type of events. This challenge is sometimes simplistically solved by collecting information about the surrounding environment only after the execution of a web page has terminated. By doing so, it is possible to determine if permanent damage has been caused to the guest OS. However, as it will be described later, there are situations in which attacks might not cause side-effects that are detectable.

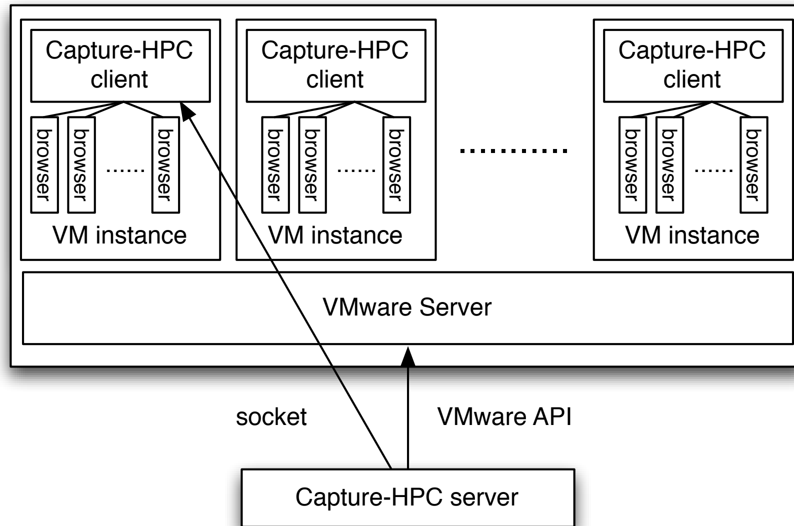

Figure 3.2: Capture-HPC Architecture

3.1.3 Honeyclients in practice

In this section, we provide a brief discussion of the general architecture and mode of operation of high-interaction honeyclients. As an example, we use Capture-HPC [99], a very popular, open-source honeyclient. To determine whether a URL is suspicious, Capture-HPC visits this URL with a browser (typically, Internet Explorer) that runs in an instrumented virtual machine.

In Figure 3.2, a more detailed overview of the architecture of Capture-HPC is shown. The system design follows a client-server model. The server component manages a number of clients and communicates with a VMware Server process to launch new, clean virtual

machine instances. Each client component is running inside one of these virtual machines. The client consists of a controller and three kernel modules to monitor file, registry, and process events, respectively. The controller receives a list of URLs from the server and opens a web browser to visit them. When a URL is visited, the kernel modules record all events that are related to the browser (by matching events against the process ID of the browser). The controller then checks the recorded events against a whitelist. This whitelist stores events that are "normal" for the execution of the browser, such as writes to the browser's cache. When events occur that are not filtered by the whitelist, the controller reports these events, together with the URL that triggered them, back to the server. Note that, for performance reasons, Capture-HPC is also capable of spawning multiple browser instances inside the same virtual machine, in order to process URLs in parallel.

In principle, all high-interaction honeyclients share an architecture that is very similar to the one described here. These systems are all based on virtual machine technology to revert to a clean state when a honeyclient instance gets compromised, and they use a client-server model to provide a URL feed and to detect suspicious activity. For example, HoneyClient [69] uses a stand-alone version of Capture-HPC's client as its detection component. Web Exploit Finder (WEF) [74] works in a way similar to Capture-HPC, but instead of using kernel modules for monitoring, the system hooks directly into the System Service Dispatch Table (SSDT) to intercept system calls. Finally, Shelia [92] takes a slightly

different monitoring approach. Instead of hooking at the kernel level, it directly hooks interesting Windows API function calls inside the browser process.

As it clear from this discussion, real-world, state-of-the-art honeyclients do not satisfy the security requirements described at the beginning of this section. First of all, they all lack transparency. All the available honeyclients operate within the guest VM, and, therefore, they can easily be detected, by looking at specific artifacts. Second, they are not tamperproof, as in a number of cases these tools can be disabled by the same malicious page that they are analyzing. Third, real-world honeyclients do not provide complete mediation. There are situations in which certain operations performed by the malicious web page can have a negative impact on the security of the browser's environment but are not monitored by the system.

Attacks that explicitly exploit these weaknesses are the subject of the next section.

3.2 Attacks Against Honeyclients

In this section, we describe techniques that an attacker can mount against a honeyclient system. The ultimate goal of the attacker is to avoid the detection of malicious activity while her URL is visited. This can be done by in two ways: *(i)* the attack code can detect the presence of the monitoring system and perform only benign activities, or *(ii)* the attack code is run in such a way that it will avoid detection. That is, the attack does not generate

any events that are considered malicious (*e.g.*, no new processes are spawned and no suspicious files or registry keys are accessed). We implemented several attacks and tested their effectiveness against the four popular honeyclients described in the previous section: Capture-HPC, HoneyClient, Shelia, and WEF. For this test, we selected a buffer overflow exploit [20] that is served in a drive-by-download via Metasploit. The victim images (the honeypots) were running Windows XP with Service Pack 2 and a vulnerable version of Internet Explorer 7. We first verified that each of the four honeypots correctly detect the attack. Then, we modified the drive-by exploit to implement each of the evasion attacks that are described below, and we checked whether the technique was successful in preventing (or evading) detection.

3.2.1 Fingerprinting the environment

A honeyclient system should not be differentiable from a true victim. If the attacker is able to detect the presence of a monitoring system, then the malicious code will compromise true victims but will perform only benign actions inside the honeypot.

Virtual machine detection. Due to the fact that many real-world drive-by attacks install malware that tampers with the operating system, most honeyclients make use of virtual machines. This way, a honeyclient can ensure that any change to the system will be reverted back to a safe state through the use of virtual machine snapshots. Although virtual machines make a honeyclient system immune to the attacks that it monitors, there is a drawback: the

attacker has the opportunity to detect the presence of the virtual machine and, when a VM is detected, conclude that the exploit code is running inside a monitored environment.

There are many known methods that can detect if code is running inside a VMware virtual machine [29, 51, 84]. By using one of these techniques, the adversary can perform a check that her code is not inside a virtualized environment. Since all the tested honeyclients make use of VMware, we have chosen the ScoopyNG checks [51] as our method to detect reliably the presence of virtual machines in practice. Although one can argue that there is an increasing tendency to use virtual machines by moving parts of applications to the cloud, we argue that regular end users do not browse the web through a VM. Rather, the normal case is to run the browser in a non-virtualized environment.

Honeyclient detection. Another way to detect the presence of a monitoring system is to check for artifacts of the honeyclient itself. For example, the attacker can detect the presence of Capture-HPC and HoneyClient by looking in the file system for the *CaptureClient.exe* binary, by checking whether specific kernel modules are loaded, or by looking for the active processes that belong to these honey clients. Similarly, Shelia and WEF can be detected by looking for their respective executable files, *shelia.exe* and *BrowserControl.exe*, or for their required DLL files. In all cases, the systems do not take any measures to protect themselves from this kind of detection.

Checking for the presence of particular files can even be done in the JavaScript code that is responsible for launching an attack. This makes the check stealthier, because no actual

```
1 var script = document.createElement("script")
2 script.type = "text/javascript";
3 script.src = "res://C:\\Program%20Files\\Capture\\CaptureClient.↩
      exe";
4 script.onreadystatechange= function () {
5   if(this.readyState == "complete")
6     alert("capture-hpc detected!"); };
7 document.getElementsByTagName("head")[0].appendChild(script);
```

Figure 3.3: Detecting Capture-HPC presence in the file system with JavaScript.

exploit is launched. In Figure 3.3, we show how an attacker can perform a simple check to see if there are specific programs installed on the client's machine. This novel attack method tricks the JavaScript engine by asking it to load a source script from the client's local file system. We found out, much to our surprise, that JavaScript does not perform any checks to see if the file requested is actually a JavaScript source file. Instead, it attempts to load the file, and, if it is not of an expected format, a JavaScript runtime error is thrown. This allows us to determine if any local file exists, simply by loading it. In this way, we are also able to detect the presence of *VMware Tools*, which reveals the existence of a virtual machine. Notice that this was tested only with Internet Explorer 7 and might not work with all of its versions.

Detection of hooked functions. Recently, there has been some effort in the research community to detect hooks installed by malware [108]. Along similar lines, we try to

detect the hooks set up by the monitoring environment. Certain honeyclients (and Shelia in particular) use function call hooking to monitor and intercept calls to critical functions. In this way, the honeyclient can prevent malicious behavior, in addition to detecting the attack. For example, the honeyclient may avoid calling the real *WinExec* function to prevent malware from executing on the system.

To hook functions, honeyclients can make use of the fact that the Windows compiler reserves the first two bytes of library functions for hot-patching. More precisely, the compiler places the instruction *MOV EDI,EDI* at the beginning of each library function prologue, which acts as a two-byte long *NOP* operation. Monitoring systems such as Shelia can then replace this instruction with a jump to a routine of their choice, which, once done, calls the original function properly. In this way, calls to critical functions such as *VirtualProtect, WinExec, etc.* can be intercepted and examined.

In Figure 3.4, we present the x86 assembly code that can be used to detect the presence of hooks before calling a function. To do this, we verify, before calling a critical function, that the first operation at the memory address where the function is located (*EBX* in our example) contains a *MOV* instruction and not *JMP* or *CALL*. As a result, the exploit code can refuse to run when function hooking is identified, or the attack code could jump over the hook to the first "real" instruction. This technique allows us to successfully detect and evade Shelia. However, this technique does not work against the other tested honeyclients, since they collect information inside the kernel.

```
checkhooks:
  CMP BYTE [DS:EBX], 0xE9     ; 0xE9 == jmp
  JE hooked
  CMP BYTE [DS:EBX], 0xE8     ; 0xE8 == call
  JE hooked
  CMP BYTE [DS:EBX], 0x8B     ; 0x8B == mov
  JE safe_vprotect
safe_vprotect:
  PUSH ESP    ; PDWORD lpflOldProtect
  PUSH 0x40   ; DWORD flNewProtect,
              ; PAGE_EXECUTE_READWRITE
  PUSH 0x7d0 ; SIZE_T dwSize , 2000
  PUSH EAX    ; LPVOID lpAddress
  CALL EBX    ; call VirtualProtect
hooked:
  ;function is hooked
  RET
```

Figure 3.4: Function hooks detection: before calling a critical function, we check if it is hooked.

HTTP referrer check. As part of a drive-by download attack, the victim issues an HTTP GET request to the attacker's web server to actually download the exploit code. Of course, the attacker is able to check the referrer of this request. If the attacker has any control

over (or knowledge about) the site(s) where her malicious URLs are being advertised, then this information can be used to serve malicious content only when the *Referer* header matches one of these expected sites. For example, the exploit might be served only if the referrer is `google.com`. Other *Referer* header values indicate that the requests are from analysis systems such as honeypots (although these systems might be able to set the *Referer* field to an expected value and thwart this check). This attack is effective against all tested honeyclients, since it occurs before the victim's infection and thus is independent of the detection approach.

3.2.2 Evading detection

Although the attacker may be unaware that her malware is running in a monitored system, she can perform some tasks that are not easily detectable by honeyclients. These tasks constitute a barrier that can be used to hide or delay the intended, malicious behavior until the adversary is sure that the attacked system is an actual victim computer.

Delayed exploitation. In a drive-by download, the attacker can decide to wait for some time before performing the exploit, or to trigger the execution of an exploit based on the occurrence of a user-triggered event. For example, the malicious JavaScript code can sleep for some seconds. The idea is that this delay will trigger a time-out on the honeyclient system (after which the analysis of the current page is terminated), but it will enable attacks against a real user who keeps the malicious page open long enough (until the time-out expires). The

honeyclients that we examined use a timeout of no more than 30 seconds, and hence, can be evaded when running the exploit is delayed by at least this amount of time.

Unfortunately, setting long timeouts makes the attack unreliable, since real users might leave the malicious site before the attack is invoked. Another way to implement a timebomb consists of waiting for some user interaction before launching the attack. This technique can be implemented by using JavaScript event handlers, such as `onMouseOver`. In this case, the idea is that a real user will likely move the mouse over the page, and, by doing so, will generate the `mouseOver` event. This will execute the corresponding handler, which, in turn, launches the attack. A honeyclient, on the other hand, will not trigger such events, and, thus, the attack remains undetected.

In-memory execution. Current honeyclient systems focus on monitoring events that are related to file system, registry, and process activity. Thus, an attack remains undetected as long as the malicious code does not interfere with these operating system resources. However, as the adversary, we would still like to execute additional malware code.

To load and execute malicious code in a stealth fashion, we can make use of remote library injection, in particular, a technique called Reflective DLL injection [30]. In this case, a (remote) library is loaded from the shellcode directly into the memory of the running process, without being registered in the process' list of loaded modules, which is stored in the Process Environment Block (PEB). Once the library is loaded, the shellcode calls an initialization function, which, in our case, injects a thread to the browser's process. At this point, the

execution is returned back to the browser, which continues to run normally. However, there is now an additional thread running that executes the malicious code.

When injecting the malicious code directly into the process, there are no additional processes spawned, files created, or registry entries manipulated. Thus, the attack evades the tested honeyclients. Of course, the malware code itself cannot write to the file system either (or it would be detect). However, it is possible to open network connections and participate in a botnet that, for example, sends spam, steals browser credentials, or conducts denial of service attacks. A drawback is that the malicious code does not survive reboots (or even closing the browser).

Whitelist manipulation. When visiting any URL, the browser interacts with the operating system and generates a certain number of events. Of course, these events do not indicate malicious behavior, and thus, they need to be removed before analyzing the effects that visiting a page has on the system. To this end, honeyclients use whitelists. However, this also means that the attacker has limited freedom in performing certain, whitelisted (operating system) actions, such as browser cache file writes, registry keys accesses *etc.*, that will not be detected as malicious. The interesting question is whether these actions can be leveraged to compromise the host's security.

The attacks described in this section are not relevant for Shelia, which uses function hooks to identify malicious activity, but apply to the remaining three honeyclients that record and monitor system calls.

```
1  void CrawlDirs(wchar startupdir[]) {
2    WIN32_FIND_DATA ffd;
3    HANDLE hFind;
4    hFind = FindFirstFile(startupdir, &ffd);
5    do {
6      if (ffd.dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY)
7        CrawlDirectories(startupdir+"\\"+ffd.cFileName);
8      else {
9        if(is_js_file(ffd.cFileName))
10         patch_js(ffd, path);
11     }
12   } while (FindNextFile(hFind, &ffd) != 0);
13 }
```

Figure 3.5: Browser cache poisoning attack.

To show the weakness of whitelisting, we have implemented a *browser cache poisoning* attack. This attack leverages that fact that write to files in the Internet Explorer cache are typically permitted. The reason is that reads and writes to these files occur as part of the normal browser operation, and hence, have to be permitted (note that Honeyclients could also disable the browser cache, making this attack ineffective).

We have implemented an attack that poisons any JavaScript file found in Internet Explorer's cache. With "poisoning," we mean that we add to every JavaScript file in the browser's

cache a small code snippet that redirects the browser to a malicious site. Thus, whenever the browser opens a page that it has visited before, and this page contains a link to a cached script, the browser will load and use the local, *modified* version of the script. As a result, the browser will get redirected and re-infected. The purpose of this attack is that it allows the adversary to make a compromise persistent *without* triggering the detection of a high-interaction honeyclient. That is, an adversary could launch an in-memory attack (as described in the previous section) and poison the cached JavaScript files with a redirect to her exploit site. Even when the victim closes the browser or reboots, it is enough to visit any page that loads a modified, cached script, to re-infect the machine in a way that is not detected by a honeyclient.

In Figure 3.5, we present a simplified version of our implementation of the cache poisoning attack. The algorithm starts from a predefined directory location (in our implementation, the directory *Temporary Internet Files*) and recursively searches for JavaScript source files. When a JavaScript source file is found, then the file is patched by inserting a redirection to the malicious site, using JavaScript's *window.location* property.

As a proof of concept for an attack that uses both in-memory execution and whitelist manipulation, we developed a keylogger that can survive boots. The keylogger runs entirely in memory, and, instead of writing the pressed keys into a file, it uses GET requests to send collected data directly to a web server.

```
1 void keylogger() {
2   wchar_t buffer[SIZE];
3   while(1) {
4     /* Appends keystrokes to buffer using GetAsyncKeyState */
5     buffer = get_keys();
6     /* Contacts attacker's webserver with buffer appended to ↩
         path requested using WinHttpSendRequest*/
7     httpget(buffer);
8   }
9 }
```

Figure 3.6: In-memory keylogger: collects keystrokes and sends them to the attacker with HTTP GET requests.

The outline of our implementation is presented in Figure 3.6. The code shows the body of the thread that is injected into *Internet Explorer's* process with the use of the Reflective DLL injection technique. The implementation is straightforward: we gather keystrokes by invoking the *GetAsyncKeyState* function offered by the Windows API. When our buffer is full, we send the keystrokes to our webserver by appending the buffer to the path field. Our keylogger is part of *Internet Explorer's* process, and thus, is very hard to detect, as it is normal for this process to perform HTTP GET requests.

To survive reboots, the keylogger also poisons all JavaScript source files in the browser cache. As a consequence, after reboot, the next time the victim visits a URL with cached JavaScript code, she will be re-infected. The honeyclients raise no alert, since all activity appears legitimate to their detection routines.

Honeyclient confusion. For performance reasons, honeyclients are capable of visiting multiple URLs at the same time. This speeds up the analysis process significantly, since the checking of URLs can be done in parallel by multiple browser instances. By using the process IDs of the different browsers, their events can be distinguished from each another.

The adversary can take advantage of this feature and try to confuse the honeyclient. In particular, the malicious code might carry out activities that are properly detected by the honeyclient as malicious, but they are blamed on a (benign) URL that is concurrently examined.

This is done by searching for concurrent, active Internet Explorer processes, as shown in Figure 3.7. Through the *IWebBrowser2* interface, we can control each browser instance, in the same way as, for example, Capture-HPC does. At this point, we can force any browser instance to visit a URL of our choice. For example, we can force the browser to visit a malicious URL under our control. This malicious URL can serve a drive-by download exploit that, when successful, downloads and executes malware. Of course, the honeyclient does not know that the browser has been forced to a different URL (by code in another browser instance), since this could also have been the effect of a benign redirect. Thus, even

when the malware performs actions that are detected, they will be blamed on the original, benign URL that Capture-HPC has initially loaded into the misdirected browser.

The purpose of this attack is to invalidate the correctness of the results produced by a honeyclient and thus, we propose to use it only when we have previously identified the presence of a monitoring system. Also, the attack does not work when a honeyclient uses only a single browser instance. However, constraining a honeyclient to test one URL at the time forces the honeyclient system to accept a major performance penalty.

3.2.3 Summary

We have implemented all the previously-described attacks and tested them against four popular, open-source honeyclients. Table 3.1 summarizes our results and shows that each honeyclient is vulnerable to most of the proposed attacks. Moreover, different attack vectors are independent and, hence, can be easily combined.

3.3 Attacks in the Real World

To better understand the extent to which high-interaction honeyclients are attacked in the real-world, we have deployed an installation of Capture-HPC. Then, we have fed this popular, high-interaction honeyclient with 33,557 URLs that were collected from various sources, such as spam URLs, web crawls, and submissions to Wepawet [18].

	Attack successful?			
Attack	Capture-HPC	Shelia	WEF	HoneyClient
Plain drive-by	✗	✗	✗	✗
VM detection	✓	✓	✓	✓
JavaScript FS checks	✓	✓	✓	✓
Hooks detection	✗	✓	✗	✗
HTTP referrer	✓	✓	✓	✓
JS timebomb	✓	✓	✓	✓
In-memory execution	✓	✓	✓	✓
Whitelist manipulation	✓	✗	✓	✓
Confusion attack	✓	✗	✓	✓

Table 3.1: Summary of the attacks: a ✗ indicates that the attack did not evade the honeyclient, a ✓ indicates that the attack was not detected.

Detection system	Total URLs	Malicious	Benign
Capture-HPC	33,557	644	32,913
Wepawet	33,557	9,230	24,327

Table 3.2: Capture-HPC and Wepawet analysis results.

Then, we compared the detection results of Wepawet and Capture-HPC for the collected URLs. Wepawet is a tool, developed by our group, that uses anomaly-based detection to identify malicious web pages by looking directly for malicious JavaScript, without checking

Malicious/Suspicious URLs undetected by Capture-HPC	
JS Method	Occurrences
setTimeOut	347
onMouseOver	419
onmouseout	403
onClick	137
Referrer	1,894

Table 3.3: Possible JavaScript evasion techniques against Capture-HPC found in the wild.

for the byproducts of a successful attack. Notice that a page marked by Wepawet as malicious contains some type of an attack that could compromise a system, but not every system will get compromised by executing the code. We have found that Wepawet has very low false positive and negative rates, and hence, its output serves as ground truth for the purpose of this evaluation [18]. Looking at Table 3.2, we can see that Wepawet found significantly more malicious sites in the wild. Of particular interest are 8,835 cases in which Wepawet detected malicious or suspicious activity, while Capture-HPC marked the URLs as benign, because there is a significant chance that these attacks try to avoid detection by systems such as Capture-HPC.

As a first step, we searched the pages that correspond to the malicious URLs for VM detection methods that are commonly employed by shellcode. Since the shellcode is typically encoded and, in addition, often obfuscated, we performed this search on the decoded JavaScript output provided by Wepawet. Moreover, we attempted to "break" the possible obfuscation of shellcode by xor'ing all shellcode strings with all possible (256) one-byte values and checking for a regular expression that indicates a (malware download) URL. Unfortunately, despite these efforts, we could not identify any shellcode that used known VM detection code.

In the next step, we checked for common JavaScript evasion techniques (as described in Section 3.2). In this case, our search turned out to be more fruitful, and we found a number of instances in which JavaScript code makes use of timeouts and event handlers (as shown in Table 3.3). Notice that the dominating technique is the HTTP *Referer* check.

We also found an interesting case in which, after the victim got infected, the malware rebooted the virtual machine. Although the attacker probably did not have the intention to avoid detection from Capture-HPC, she actually did. In fact, since the machine is rebooting, all running processes are killed. This means that Capture-HPC's client is also killed. This happens immediately after infection. Because Capture-HPC's client waits for all browser instances to finish loading their URLs, this event precedes reporting back to Capture-HPC's server. As a result, the server loses the connection with the client. When this happens, the server simply believes that the client has crashed, so it restores the virtual machine to a safe

snapshot and restarts the client. This means that all the information gathered by the previous instance of the client is lost. Thus, although the virtual machine is rebooting, an obvious malicious act, Capture-HPC is unable to detect the malicious URL that caused it.

Another (possibly involuntary) evasion technique that we observed has to do with the fact that Capture-HPC is built as a Windows Console Application. After the infection, we noticed that a new command (DOS) window was spawned by the exploit code. After a few seconds, this command prompt was killed. But, by doing so, the Capture-HPC process was killed as well. The reason is that Capture-HPC is also invoked from the Windows command prompt (*cmd.exe*). Thus, when the exploit code kills all command prompts as part of its operations, the Capture-HPC process is killed as well. From this point on, the detection fails for the same reasons described in the previous paragraph.

3.4 Countermeasures

In this chapter, we have shown a number of attacks that an adversary can use to prevent the detection of her malicious web pages by a honeyclient system. In this section, we focus on the countermeasures that existing honeyclient systems may implement to detect attacks more robustly.

3.4.1 Transparency

We have seen that it is often easy for a malicious page to detect that it is running in a high-interaction honeyclient (as opposed to a regular host). This issue should be addressed at several levels.

File system level. The attacker should not be able to discover the files related to the monitoring infrastructure on the honeyclient's file system. For example, in the default setup of Capture-HPC, a simple search into the *Program Files* directory reveals the use of VMware Tools and Capture-HPC. These files should be hidden in the file system so that the attacker must put significant effort in finding them, and the honeyclient should be able to detect the attacker's attempt to locate these files.

Kernel level. The attacker should not be able to detect the presence of kernel drivers that honeyclients use for monitoring system level changes. We are not aware of any method offered by the Windows API that can be used to hide the presence of a driver from a process. Therefore, honeyclients could rely on rootkit techniques, such as Direct Kernel Object Manipulation (DKOM) [101], to hide the presence of these modules. Notice that, by using the same techniques, a honeyclient could hide processes and files that it uses for its monitoring.

Network level. If the monitored system is deployed on a static location on the Internet, its IP address or subnet may be identified by attackers [53]. It is, therefore, necessary to

deploy the monitored system in several locations, using multiple ISPs in order eliminate the possibility of getting blacklisted by the attacker at the network level.

Hypervisor level. Implementing the monitoring system at the hypervisor level offers complete isolation between the analysis environment and the malware. Although this approach seems ideal, inspecting the operating system from "outside the box" is not trivial, and it requires a significant effort to reverse engineer the necessary operating system data structures to bridge the semantic gap.

Thwarting virtual machine detection. The virtual machines currently used for malicious behavior analysis are not designed to be transparent [33]. As we have seen in Chapter 2, there has been significant effort to create stealthier virtual machines, such as MAVMM [76], and transparent monitoring systems, such as Ether [23]. These techniques could be used in future honeyclient systems.

3.4.2 Protection of the monitoring system

Protecting the browser. A successful exploit against a browser vulnerability typically gives the attacker the ability to execute arbitrary code in the context of the exploited browser process. The attacker can then subvert other browser processes, compromising the integrity of the detection, as we have seen in the case of the confusion attack. High-interaction honeyclients that run multiple browser instances should take steps to isolate each instance from the others, for example by executing them under different principals. Alternatively,

the honeyclient process could monitor browser processes to detect attempts to manipulate their execution. For example, the honeyclient system could monitor the *Handles* that belong to each browser's process, using the *GetProcessHandleCount* function provided by the Windows API. In this fashion, one can monitor for cases when the attacker attempts to manipulate a browser and protect the results produced by revisiting one by one the URLs associated with the manipulated browser's instances.

Protecting the honeyclient processes. Any honeyclient process that runs inside the virtual machine needs to be protected from tampering (e.g., from getting terminated) by the attacker. One way to achieve this is by running the honeyclient processes with elevated privileges compared to the browser's processes. It is also possible to check for and intercept attempts to terminate the honeyclient processes.

3.5 Conclusion

In this chapter, we examined the security model that high-interaction honeyclients use, and we evaluated their weaknesses in practice. We introduced and discussed a number of possible attacks, and we test them against several popular, well-known high-interaction honeyclients. In particular, we have introduced three novel attack techniques (JavaScript-based honeyclient detection, in-memory execution, and whitelist-based attacks) and put under the microscope already-known attacks. Our attacks evade the detection of the

tested honeyclients, while successfully compromising regular visitors. Furthermore, we suggest several countermeasures aiming to improve honeyclients. By employing these countermeasures, a honeyclient will be better protected from evasion attempts and will provide more accurate results.

```
1  SHDocVw::IShellWindowsPtr spSHWinds;
2  IDispatchPtr spDisp;
3  IWebBrowser2 * pWebBrowser = NULL;
4  HRESULT hr;
5
6  // get all active browsers
7  spSHWinds.CreateInstance(__uuidof(SHDocVw::ShellWindows));
8
9  // get one, or iterate va to get each one
10 spDisp = spSHWinds->Item (va);
11
12 // get IWebBrowser2 pointer
13 hr = spDisp.QueryInterface (IID_IWebBrowser2, & pWebBrowser);
14
15 if (SUCCEEDED(hr) && pWebBrowser != NULL) {
16   visitUrl(pWebBrowser); // with the use of IWebBrowser2::↩
         Navigate2
17 }
```

Figure 3.7: Confuse honeyclient: find an Internet Explorer instance and force it to visit a URL of our choice.

Chapter 4

An Automated Approach to the Detection of Evasive Web-based Malware

In the previous chapter we showed how evasions work against high-interaction honeyclients. In this chapter, we focus on JavaScript-based evasions that target low-interaction honeyclients by introducing *Revolver*, a novel approach to automatically detect evasive behavior in malicious JavaScript. *Revolver* uses efficient techniques to identify similarities between a large number of JavaScript programs (despite their use of obfuscation techniques, such as packing, polymorphism, and dynamic code generation), and to automatically interpret their differences to detect evasions.

In the last several years, we have seen web-based malware—malware distributed over the web, exploiting vulnerabilities in web browsers and their plugins—becoming a prevalent threat. Microsoft reports that it detected web-based exploits against over 3.5 million distinct computers in the first quarter of 2012 alone [68]. In particular, drive-by-download

attacks are the method of choice for attackers to compromise and take control of victim machines [39, 82]. At the core of these attacks are pieces of malicious HTML and JavaScript code that launch browser exploits.

Recently, a number of techniques have been proposed to detect the code used in drive-by-download attacks. A common approach is the use of honeyclients (specially instrumented browsers) that visit a suspect page and extract a number of features that help in determining if a page is benign or malicious. Such features can be based on static characteristics of the examined code [14, 19], on specifics of its dynamic behavior [18, 61, 75, 81, 88, 106], or on a combination of static and dynamic features [91].

Drive-by downloads initially contained only the code that exploits the browser. This approach was defeated by static detection of the malicious code using signatures. The attackers started to obfuscate the code in order to make the attacks impossible to be matched by signatures. Obfuscated code needs to be executed by a JavaScript engine to truly reveal the final code that performs the attack. This is why researchers moved to dynamic analysis systems which execute the JavaScript code, deobfuscating this way the attacks regardless of the targeted vulnerable browser or plugin. As a result, the attackers have introduced evasions: JavaScript code that detects the presence of the monitoring system and behaves differently at runtime. Any diversion from the original targeted vulnerable browser (e.g., missing functionality, additional objects, etc.) can be used as an evasion.

As a result, malicious code is not a static artifact that, after being created, is reused without changes. To the contrary, attackers have strong motivations to modify the code they use so that it is more likely to evade the defense mechanisms employed by end-users and security researchers, while continuing to be successful at exploiting vulnerable browsers. For example, attackers may obfuscate their code so that it does not match the string signatures used by antivirus tools (a situation similar to the polymorphic techniques used in binary malware). Attackers may also mutate their code with the intent of evading a specific detection tool, such as one of the honeyclients mentioned above.

In this chapter we propose *Revolver*, a novel approach to automatically identify evasions in drive-by-download attacks. In particular, given a piece of JavaScript code, *Revolver* efficiently identifies scripts that are *similar* to that code, and automatically classifies the differences between two scripts that have been determined to be similar. *Revolver* first identifies syntactic-level differences in similar scripts (e.g., insertion, removal, or substitution of snippets of code). Then *Revolver* attempts to explain the semantics of such differences (i.e., their effect on page execution). We show that these changes often correspond to the introduction of evasive behavior (i.e., functionality designed to evade popular honeyclient tools).

There are several challenges that *Revolver* needs to address to make this approach feasible in practice. First, typical drive-by-download web pages serve malicious code that is heavily obfuscated. The code may be mutated from one visit to the page to the next by using

simple polymorphic techniques, e.g., by randomly renaming variables and functions names. Polymorphism creates a multitude of differences in two pieces of code. From a superficial analysis, two functionally identical pieces of code will appear as very different. In addition, malicious code may be produced on-the-fly, by dynamically generating and executing new code (through JavaScript and browser DOM constructs such as the `eval()` and `setTimeout()` functions). Dynamic code generation poses a problem of coverage; that is, not all JavaScript code may be readily available to the analyzer. Therefore, a naive approach that attempts to directly compare two malicious scripts would be easily thwarted by these obfuscation techniques and would fail to detect their similarities. Instead, *Revolver* dynamically monitors the execution of JavaScript code in a web page so that it can analyze both the scripts that are statically present in the page and those that are generated at runtime. In addition, to overcome polymorphic mutations of code, *Revolver* performs its similarity matching by analyzing the Abstract Syntax Tree (AST) of code, thereby ignoring superficial changes to its source code.

Another challenge that *Revolver* must address is scalability. For a typical analysis of a web page, *Revolver* needs to compare several JavaScript scripts (more precisely, their ASTs) with a repository of millions of ASTs (potential matches) to identify similar ones. To make this similarity matching computationally efficient, we use a number of machine learning techniques, such as dimensionality reduction and clustering algorithms.

Finally, not all code changes are security-relevant. For example, a change in a portion of the code that is never executed is less interesting than one that causes a difference in the runtime behavior of the script. In particular, we are interested in identifying code changes that cause detection tools to misclassify a malicious script as benign. To identify such evasive code changes, *Revolver* focuses on modifications that introduce control flow changes in the program. These changes may indicate that the modified program checks whether it is being analyzed by a detector tool (rather than an unsuspecting visitor) and exhibits a different behavior depending on the result of this check.

By automatically identifying code changes designed to evade drive-by-download detectors, one can improve detection tools and increase their detection rate. We also leverage *Revolver* to identify benign scripts (e.g., well-known libraries) that have been injected with malicious code, and, thus, display malicious behavior.

This chapter makes the following contributions:

- **Code similarity detection**: We introduce techniques to efficiently identify JavaScript code snippets that are similar to each other. Our tool is resilient to obfuscation techniques, such as polymorphism and dynamic code generation, and also pinpoints the precise differences (changes in their ASTs) between two different versions of similar scripts.
- **Detection of evasive code**: We present several techniques to automatically classify differences between two similar scripts to highlight their purpose and effect on the

executed code. In particular, *Revolver* has identified several techniques that attackers use to evade existing detection tools by continuously running in parallel with a honeyclient.

4.1 Background and Overview

To give the reader a better understanding of the motivation for our system and the problems that it addresses, we start with a discussion of malicious JavaScript code used in drive-by-download attacks. Moreover, we present an example of the kind of code similarities that we found in the wild.

Malicious JavaScript code. The web pages involved in drive-by-download attacks typically include malicious JavaScript code. This code is usually obfuscated, and it fingerprints the visitor's browser, identifies vulnerabilities in the browser itself or the plugins that the browser uses, and finally launches one or more exploits. These attacks target memory corruption vulnerabilities or insecure APIs that, if successfully exploited, enable the attackers to execute arbitrary code of their choice.

Figure 4.1 shows a portion of the code used in a recent drive-by-download attack against users of the Internet Explorer browser. The code (slightly edited for the sake of clarity) instantiates a shellcode (Line 8) by concatenating the variables defined at Lines 1–7; a later portion of the code (not shown in the figure) triggers a memory corruption vulnerability, which, if successful, causes the shellcode to be executed.

A common approach to detect such attacks is to use honeyclients, which are tools that pose as regular browsers, but are able to analyze the code included in the page and the side-effects of its execution. More precisely, low-interaction honeyclients emulate regular browsers and use various heuristics to identify malicious behavior during the visit of a web page [18,41,75]. High-interaction honeyclients consist of full-featured web browsers running in a monitoring environment that tracks all modifications to the underlying system, such as files created and processes launched [81, 99, 106]. If any unexpected modification occurs, it is considered to be a manifestation of a successful exploit. Notice that this sample is difficult to detect with a signature, as strings are randomized on each visit to the compromised site.

Evasive code. Attackers have a vested interest in crafting their code to evade the detection of analysis tools, while remaining effective at exploiting regular users. This allows their pages to stay "under the radar" (and actively malicious) for a longer period of time, by avoiding being included in blacklists such as Google's Safe Browsing [37] or being targeted by take-down requests.

Attackers can use a number of techniques to avoid detection [86]: for example, code obfuscation is effective against tools that rely on signatures, such as antivirus scanners; requiring arbitrary user interaction can undermine high-interaction honeyclients; probing for arcane characteristics of browser features (likely not correctly emulated in browser emulators) can thwart low-interaction honeyclients.

```
1 var nop="%uyt9yt2yt9yt2";
2 var nop=(nop.replace(/yt/g,""));
3 var sc0="%ud5db%uc9c9%u87cd...";
4 var sc1="%"+"yutianu"+"ByutianD"+ ...;
5 var sc1=(sc1.replace(/yutian/g,""));
6 var sc2="%"+"u"+"54"+"FF"+...+"8"+"E"+"E";
7 var sc2=(sc2.replace(/yutian/g,""));
8 var sc=unescape(nop+sc0+sc1+sc2);
```

Figure 4.1: Malicious code that sets up a shellcode.

An effective way to implement this kind of circumventing techniques consists of adding some specialized “evasive code” whose only purpose is to cause detector tools to fail on an existing malicious script. Of course, the evasive code is designed in such a way that regular browsers (used by victims) effectively ignore it. Such evasive code could, for example, pack an exploit code in an obfuscation routine, check for human interaction, or implement a test for detecting browser emulators (such evasive code is conceptually similar to “red pills” employed in binary malware to detect and evade commonly-used analysis tools [29]).

Figure 4.2 shows an evasive modification to the original exploit of Figure 4.1, which we also found used in the wild. More precisely, the code tries to load a non-existent ActiveX control, named `yutian` (Line 2). On a regular browser, this operation fails, triggering the execution of the catch branch (Lines 4–11), which contains an identical copy of the malicious code of Figure 4.1. However, low-interaction honeyclients usually emulate the ActiveX

```
1  try {
2      new ActiveXObject("yutian");
3  } catch (e) {
4      var nop="%uyt9yt2yt9yt2";
5      var nop=(nop.replace(/yt/g,""));
6      var sc0="%ud5db%uc9c9%u87cd...";
7      var sc1="%"+"yutianu"+"ByutianD"+ ...;
8      var sc1=(sc1.replace(/yutian/g,""));
9      var sc2="%"+"u"+"54"+"FF"+...+"8"+"E"+"E";
10     var sc2=(sc2.replace(/yutian/g,""));
11     var sc=unescape(nop+sc0+sc1+sc2);
12 }
```

Figure 4.2: An evasion using non-existent ActiveX controls.

API by simulating the presence of *any* ActiveX control. In these systems, the loading of the ActiveX control does not raise any exception; as a consequence, the shellcode is not instantiated correctly, which stops the execution of the exploits and causes the honeyclient to fail to detect the malicious activity.

Detecting evasive code using code similarity. Code similarity approaches have been proposed in the past, but none of them has focused specifically on malicious JavaScript. There are several challenges involved when processing malicious JavaScript for similarities. Attackers actively try to trigger parsing issues in analyzers. The code is usually heavily obfuscated, which means that statically examining the code is not enough. The malicious

code itself is designed to evade signature detection from antivirus products. This renders string-based and token-based code similarity approaches ineffective against malicious JavaScript. We will show later how regular code similarity tools, such as Moss [96], fail when analyzing obfuscated scripts. In *Revolver*, we extend tree-based code similarity approaches and focus on making our system robust against malicious JavaScript. We elaborate on our novel code similarity techniques in §4.2.4.

At a high-level overview, we use *Revolver* to detect and understand the similarity between two code scripts. Intuitively, *Revolver* is provided with the code of both scripts and their classification by one or more honeyclient tools. In our running example, we assume that the code in Figure 4.1 is flagged as malicious and the one in Figure 4.2 as benign. *Revolver* starts by extracting the Abstract Syntax Tree (AST) corresponding to each script. *Revolver* inspects the ASTs rather than the original code samples to abstract away possible superficial differences in the scripts (e.g., the renaming of variables). When analyzing the AST of Figure 4.2, it detects that it is similar to the AST of the code in Figure 4.1. The change is deemed to be interesting, since it introduces a difference (the try-catch statement) that may cause a change in the control flow of the original program. Our system also determines that the added code (the statement that tries to load the ActiveX control) is indeed executed by tools visiting the page, thus increasing the relevance of the detected change (*execution bits* are described in more detail in §4.2.1). Finally, *Revolver* classifies the modification as a

possible evasion attempt, since it causes the honeyclient to change its detection result (from malicious to benign).

Assumptions and limitations. Our approach is based on a few assumptions. *Revolver* relies on external detection tools to collect (and make available) a repository of JavaScript code, and to provide a classification of such code as either malicious or benign (i.e., *Revolver* is not a detection tool by itself). To obtain code samples and classification scores, we can rely on several publicly-available detectors [18, 41, 75].

Attackers might write a brand new attack with all components (evasion, obfuscation, exploit code) written from scratch. In such cases, *Revolver* will not be able to find any similarities the first time it analyzes these attacks. The lack of similarities though can be used to our advantage, since we can isolate brand-new attacks (provided that they can be identified by other means) based on the fact that we have never observed such code before.

In the same spirit, to detect evasions, *Revolver* needs to inspect two versions of a malicious script: the "regular" version, which does not contain evasive code, and the "evasive" version, which attempts to circumvent detection tools. Furthermore, if an evasion is occurring, we assume that a detection tool would classify these two versions differently. In particular, if only the evasive version of a JavaScript program is available, *Revolver* will not be able to detect this evasion. We consider this condition to be unlikely. In fact, trend results from a recent Google study on circumvention [86] suggest that malicious code evolves over time to incorporate more sophisticated techniques (including evasion). Thus, having a

sufficiently large code repository should allow us to have access to both regular and evasive versions of a script. Furthermore, we have anecdotal evidence of malware authors creating different versions of their malicious scripts and submitting them to public analyzers, until they determine that their programs are no longer detected (this situation is reminiscent of the use of anti-antivirus services in the binary malware world [54]).

Revolver is not effective when server-side evasion (for example, IP cloaking) is used: in such cases, the malicious web site does not serve at all the malicious content to a detector coming from a blocked IP address, and, therefore, no analysis of its content is possible. This is a general limitation of all analysis tools and can be solved by means of a better analysis infrastructure (for example, by visiting malicious sites from IP addresses and networks that are not known to be associated with analysts and security researchers and cannot be easily fingerprinted by attackers).

4.2 Approach

In this section, we describe *Revolver* in detail, focusing on the techniques that it uses to find similarities between JavaScript files.

A high-level overview of *Revolver* is presented in Figure 4.3. First, we leverage an existing drive-by-download detection tool (an "Oracle") to collect datasets of both benign and malicious web pages (§4.2.1). Second, *Revolver* extracts the ASTs (§4.2.2) of the

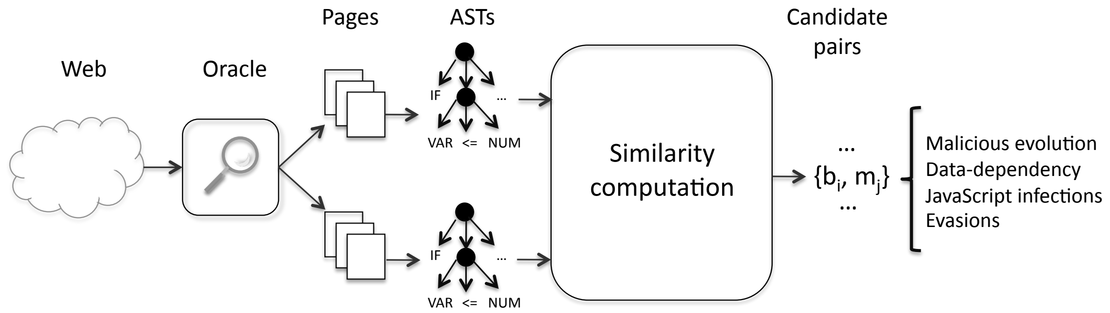

Figure 4.3: Architecture of *Revolver*.

JavaScript code contained in these pages and, leveraging the Oracle's classification for the code that contains them, marks them as either benign or malicious. Third, *Revolver* computes a similarity score for each pair of ASTs, where one AST is malicious and the other one can be either benign or malicious (§4.2.3–§4.2.4). Finally, pairs that are found to have a similarity score higher than a given threshold are further analyzed to identify and classify their similarities (§4.2.5).

If *Revolver* finds similarities between two malicious scripts, then we classify this case as an instance of *evolution* (typically, an improvement of the original malicious code). On the other hand, if *Revolver* detects similarities between a malicious and a benign script, it performs an additional classification step. In particular, similarities can be classified by *Revolver* into one of four possible categories: *evasions*, *injections*, *data dependencies*, and *general evolutions*. We are especially interested in identifying evasions, which indicate changes that cause a script that had been found to be malicious before to be flagged as benign now.

It is important to note that, due to JavaScript's ability to produce additional JavaScript code on the fly (which enables extremely complex JavaScript packers and obfuscators), performing this analysis statically would not be possible. *Revolver* works dynamically, by analyzing all JavaScript code that is compiled in the course of a web page's execution. By including all these scripts, and the relationships between them (such as what code created what other code), *Revolver* is able to calculate JavaScript similarities among malicious web pages to an extent that is not, to our knowledge, possible with existing state-of-the-art code comparison tools.

4.2.1 Oracle

Revolver relies on existing drive-by-download detection tools for a single task: the classification of scripts in web pages as either malicious or benign. Notice that our approach is not tied to a specific detection technique or tool; therefore, we use the term "Oracle" to generically refer to any such detection system. In particular, several popular low- and high-interaction honeyclients (e.g., [18, 41, 75, 99]) or any antivirus scanner can readily be used for *Revolver*.

Revolver analyzes the Abstract Syntax Trees (ASTs) of individual scripts rather than examining web pages as a whole. Therefore, *Revolver* performs a refinement step, in which i) individual ASTs are extracted from the web pages obtained from the Oracle, ii) their detection status is determined (that is, each AST is classified as either benign or malicious), based on the page classification provided by the Oracle, and iii) for each node in an AST,

it is recorded whether the corresponding statement was executed. Of course, if an Oracle natively provides this fine-grained information, this step can be skipped.

More precisely, *Revolver* executes each web page using a browser emulator based on HtmlUnit [34]. The emulator parses the page and extracts all of its JavaScript content (e.g., the content of `script` tags and the body of event handlers). In particular, the ASTs of the JavaScript code are saved for later analysis. In addition, to obtain the AST of dynamically-generated code, *Revolver* executes the JavaScript code. At the end of the execution, for each node in the AST, *Revolver* keeps an **execution bit** to record whether the code corresponding to that node was executed. Whenever it encounters a function that generates new code (e.g., a call to the `eval()` or `setTimeout()` functions), *Revolver* analyzes the code that is generated by these functions. It also saves the parent-child relationship between scripts, i.e., which script is responsible for the execution of a dynamically-generated script. For example, the script containing the `eval()` call is considered the parent of the script that is evaluated. Similarly, *Revolver* keeps track of which script causes network resources to be fetched, for example, by creating an `iframe` tag.

Second, for each AST, *Revolver* determines if it is malicious or benign, based on the Oracle's input. More precisely, an AST is considered malicious if it is the parent of a malicious AST, or if it issued a web request that led to the execution of malicious code. This makes *Revolver* flexible enough to work with any Oracle.

4.2.2 Abstract Syntax Trees

Revolver's core analysis is based on the examination of ASTs rather than the source code of a script. The rationale for using ASTs is that they abstract away details that are irrelevant for our analysis (and, in fact, often undesirable), while retaining enough precision to achieve good results.

For example, consider a script obtained from the code in Figure 4.1 via simple obfuscation techniques: renaming of variables and function names, introduction of comments, and randomization of whitespace. Clearly, we want *Revolver* to consider these scripts as similar. Making this decision can be non-trivial when inspecting the source code of the scripts. In fact, as a simple validation, we ran Moss, a system for determining the similarity of programs, which is often used as a plagiarism detection tool [96], on the original script and the one obtained via obfuscation. Moss failed to flag the two scripts as similar, as shown in the tool's output here [72]. However, the two scripts are identical when their AST representations are considered, since, in the trees, variables are represented by generic `VAR` nodes, independently of their names, and comments and whitespaces are ignored. This makes tree-based code similarity approaches more suitable for malicious JavaScript comparisons (and this is the reason why our analysis leverages ASTs as well). However, as shown in §4.2.4, we need to treat malicious code in a way that is different from previous

techniques targeting benign codebases. Below, we describe our approach and necessary extensions in more detail.

Revolver transforms the AST produced by the JavaScript compiler into a *normalized node sequence*, which is the sequence of node types obtained by performing a pre-order visit of the tree. In total, there are 88 distinct node types, corresponding to different constructs of the JavaScript language. Examples of the node types include `IF`, `WHILE`, and `ASSIGN` nodes.

Figure 4.4 summarizes the data structures used by *Revolver* during its processing. We discuss *sequence summaries* in the next Section.

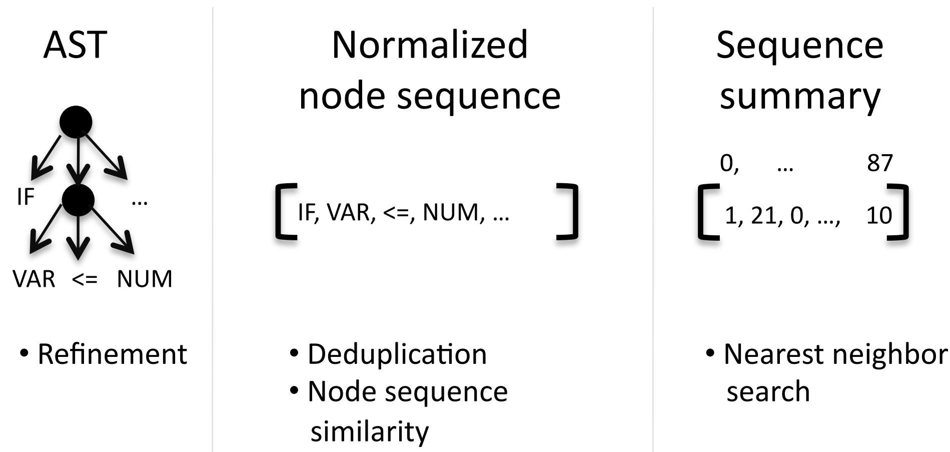

Figure 4.4: Data structures used by *Revolver*.

4.2.3 Similarity Detection

After extracting an AST and transforming it in its normalized node sequence, *Revolver* finds similar normalized node sequences. The result is a list of normalized node sequence

pairs. In particular, pairs of malicious sequences are compared to identify cases of evolution; pairs where one of the sequences is benign and the other malicious are analyzed to identify possible evasion attempts.

The similarity computation is based on computing the directed edit distance between two node sequences, which, intuitively, corresponds to the number of operations that are required to transform one benign sequence into the malicious one. Before discussing the actual similarity measurement, we discuss a number of minimization techniques that we use to make the computation of the similarity score feasible in datasets comprising millions of node sequences.

Deduplication. As a first step to reduce the number of similarity computations, we discard duplicates in our dataset of normalized node sequences. Since we use a canonical representation for the ASTs, we can easily and efficiently compute hashes of each sequence, which enables us to quickly identify groups of identical node sequences. In the remaining processing phases, we only need to consider one member of a group of identical node sequences (rather than all of its elements). Notice that identical normalized node sequences may correspond to different scripts, and may also have a different detection classification (we describe such cases in §4.2.5). Therefore, throughout this processing, we always maintain the association between node sequences and the scripts they correspond to, and whether they have been classified as malicious or benign.

Approximate nearest neighbors. Given a repository of n benign ASTs and m malicious ones, *Revolver* needs to compute $n \times m$ comparisons over (potentially long) node sequences. Even after the deduplication step, this may require a significantly large number of operations.

To address this problem, we introduce the idea of *sequence summaries*. A sequence summary is a compact summarization of a regular normalized node sequence, which stores the number of times each node type appears in the corresponding AST. Since there are 88 distinct node types, each node sequence is mapped into a point in an 88-dimensional Euclidean space. An advantage of sequence summaries is that they bound the length of the objects that will be compared (from potentially very large node sequences, corresponding to large ASTs, down to more manageable vectors of fixed length).

Then, for each sequence summary s, we identify its *malicious neighbors*, that is, up to k malicious sequence summaries t, such that the distance between s and t is less than a chosen threshold τ_n. Intuitively, the malicious neighbors correspond to the set of ASTs that we expect to be most similar to a given AST. Determining the malicious neighbors of a sequence summary is an instance of the k-nearest neighbor search problem, for which various efficient algorithms have been proposed. In particular, we solve it by using the FLANN library [73].

In the remaining step, we compare sequence summaries only with their malicious neighbors, thus dramatically reducing the number of comparison to be performed.

Normalized node sequence similarity. Finally, we can compute the similarity between two normalized node sequences. More precisely, *Revolver* compares the normalized node

sequence corresponding to a sequence summary s with each normalized node sequence that corresponds to a sequence summary of the malicious neighbors of s.

The similarity measurement is based on the pattern matching algorithm by Ratcliff et al. [89]. More precisely, given two node sequences, a and b, we first find their longest contiguous common subsequence (LCCS). Then, we recursively search for LCCS between the pieces of a and b to the left and right of the matching subsequence. The similarity of a and b is then returned as the number of nodes in common divided by the total number of nodes in the malicious node sequence. Therefore, identical ASTs will have similarity 1, and the similarity values decrease toward zero as two ASTs contain higher amounts of different nodes. This technique is robust against injections, where one benign script includes a malicious one, since all malicious nodes will be matched.

In addition to a numeric similarity score, the algorithm also provides a list of insertions for the two node sequences, that is, a list of AST nodes that would need to be added to one sequence to transform it into the other one. This information is very useful for our analysis, since it identifies the actual code that was added to an original malicious code.

After the similarity score is computed, we discard any pairs that have a similarity below a predetermined threshold τ_s.

Expansion. Once pairs of ASTs with high similarity have been identified, we need to determine the Oracle's classification of the scripts they originate from. We, therefore, expand

out any pairs that we deduplicated in the initial *Deduplication* step so that we associate the AST similarities to the scripts that they correspond to.

4.2.4 Optimizations

There are several techniques that we utilize to improve the results produced by the similarity detection steps. In particular, our objective is to restrict the pairs identified as similar to "interesting" ones, i.e., those that are more likely to correspond to evasion attempts or significant new functionality. The techniques introduced here build upon tree-based code similarity approaches and are specific to malicious JavaScript.

Size matters. We observed that JavaScript code contains a lot of very small scripts. In the extreme case, it includes scripts comprising a single statement. We determined that the majority of such scripts are generated dynamically through calls to `eval()`, which, for example, dynamically invoke a second function. Such tiny scripts are problematic for our analysis: they have not enough functionality to perform malicious actions and they end up matching other short scripts, but their similarity is not particularly relevant. As a consequence, we **combine** ASTs that contain less than a set number of nodes (τ_z). We do this by taking into account how a script was generated: if another script generated code under our threshold, we inline the generated script back to its parent. If the script was not dynamically generated, then we treat it as if one script contained all static code under our

threshold. This way the attacker cannot split the malicious code into multiple parts under our threshold in order to evade *Revolver*.

Repeated pattern detection. We also observed that, in certain cases, an AST may contain a set of nodes repeated a large number of times. This commonly occurs when the script uses some JavaScript data structure that yields many repeated AST nodes. For example, malicious scripts that unpack or deobfuscate their exploit payload frequently utilize a JavaScript *Array* of elements to store the payload. Their ASTs contain a node for every single element in the *Array*, which, in many cases, may have thousands of instances. An unwanted consequence, then, is that any script with a large *Array* will be considered similar to the malicious script (due to the high similarity of the array nodes), regardless of the presence of a decoding/unpacking routine (which, instead, is critical to determine the similarity of the scripts from a functional point of view). These obfuscation artifacts affect tree-based similarity algorithms, which will result in the detection of similar code pairs where the common parts are of no interest in the context of malicious JavaScript. To avoid this problem, we identify sequences of nodes that are repeated in a script more than a threshold (τ_p) and truncate them.

Similarity fragmentation. Although we have identified blocks of code that are shared across two scripts, it can be the case that these blocks are not continuous. One script can be broken down into small fragments that are matched to the other script in different positions. This is why we take into account the fragmentation of the matching blocks. To prune these

cases, we recognize a similarity only if the fragmentation of the similarities is below a set threshold τ_f.

4.2.5 Classification

The outcome of the previous similarity detection step is a list of pairs of scripts that are similar. As we show in §4.4.1 we can have hundreds of thousands of similar pairs. Therefore, *Revolver* performs a classification step of similar pairs. That is, *Revolver* interprets the changes that were made between two scripts and classifies them. There are two cases, depending on the Oracle's classification of the scripts in a pair. If the pair consists solely of malicious scripts, then we classify the similarity as a malicious evolution. The other case is a pair in which one script is malicious and one script is benign. We call such pairs *candidate pairs* (they need to be further tested before we can classify their differences). While the similarity detection has operated on a syntactic level (essentially, by comparing AST nodes), *Revolver* now attempts to determine the semantics of the differences.

In practice, *Revolver* classifies the scripts and their similarities into one of several categories, corresponding to different cases where an Oracle may flag differently scripts that are similar. Table 4.1 summarizes the classification algorithm used by *Revolver*.

Data-dependency category. *Revolver* checks if a pair of scripts belongs to the data-dependency category. A typical example of scripts that fall into this category is packers. Packers are tools that allow JavaScript authors to deliver their code in a packed format,

AST	Executed nodes	Classification
$=$	$*$	Data-dependency
$*$	$=$	Data-dependency
$B \subseteq M$	$\neq$	JavaScript injection
$M \subseteq B$	$\neq$	Evasion
$\neq$	$\neq$	General evolution

Table 4.1: Candidate pairs classification (B is a benign sequence, M is a malicious sequence, $*$ indicates a wildcard value).

significantly reducing the size of the code. In packed scripts, the original code of the script is stored as a compacted string or array, and its clear-text version is retrieved at run-time by executing an unpacking routine. Packers have legitimate uses (mostly, size compression): in fact, several open-source popular packers exist [26], and they are frequently used to distribute packed version of legitimate JavaScript libraries, such as jQuery. However, malware authors also rely on these very same packers to obfuscate their code and make it harder to be fingerprinted.

Notice that the ASTs of packed scripts (generated by the same packer) are identical, independently of their (unpacked) payload: in fact, they consist of the nodes of the unpacking routine (which is fixed) and of the nodes holding the packed data (typically, the assignment

of a string literal to a variable). However, the actual packed contents, which eventually determine whether the script is malicious or benign, are not retained at the AST level of the packer, but the packed content will eventually determine the nature of the overall script as benign or malicious.

Revolver categorizes as data-dependent pairs of scripts that are identical and have different detection classification.

As a slight variation to this scenario, *Revolver* also classifies as data-dependent pairs of scripts for which the ASTs are not identical, but the set of nodes that were actually executed are indeed the same. For example, this corresponds to cases where a function is added to the packer but is never actually executed during the unpacking.

Control-flow differences. The remaining categories are based on the analysis of AST nodes that are different in the two scripts, and, specifically, of nodes representing control-flow statement. We focus on such nodes because they give an attacker a natural way to implement a check designed to evade detection. In fact, such checks generally test a condition and modify the control flow depending on the result of the test.

More precisely, we consider the following control-flow related nodes: *TRY, CATCH, CALL, WHILE, FOR, IF, ELSE, HOOK, BREAK, THROW, SWITCH, CASE, CONTINUE, RETURN, LT (<), LE (<=), GT (>), GE (>=), EQ (==) , NE (! =), SHEQ (===), SNE (! ==), AND, and OR*. Depending on where these control-flow nodes were added, whether in the benign or in the malicious script, a candidate pair can be classified as a JavaScript

injection or an evasion. Notice that we leverage here the *execution bits* to detect control flow changes that were actually executed and affected the execution of code that was found as malicious before.

JavaScript injection category. In some cases, malware authors insert malicious JavaScript code into existing benign scripts on a compromised host. This is done because, when investigating a compromise, webmasters may neglect to inspect files that are familiar to them, and thus such injections can go undetected. In particular, it is common for malware authors to add their malicious scripts to the code of popular JavaScript libraries hosted on a compromised site, such as jQuery and SWFObject.

In these cases, *Revolver* identifies similarities between a benign script (the original, legitimate jQuery code) and a malicious script (the library with the added malicious code). In addition, *Revolver* detects that the difference between the two scripts is due to the presence of control-flow nodes in the malicious script (the additional code added to the library), which are missing in the benign script. *Revolver* classifies such similarities as JavaScript injections, since the classification of the analyzed script changes from benign to malicious due to the introduction of additional code in the malicious version of the script.

Evasions category. Pairs of scripts that only differ because of the presence of additional control-flow nodes in the benign script are categorized as evasions. In fact, these correspond to cases where a script, which was originally flagged as malicious by an Oracle, is modified

to include some additional functionality that modifies its control flow (i.e., an evasive check) and, as a consequence, appears to be benign to the Oracle.

General evolution cases. Finally, if none of the previous categories applies to the current pair of scripts, it means that their differences are caused by the insertion of control-flow nodes in both the benign and malicious scripts. Unlike similarities in the evasion category, these similarities may signify generic evolution between the two scripts. *Revolver* flags these cases for manual review, at a lower priority than evasive cases.

4.3 Implementation

In this section, we discuss specific implementation choices for our approach.

We used the Wepawet honeyclient [18] as the Oracle of *Revolver*. In particular, the input to *Revolver* was the web pages processed by the Wepawet tool at real-time together with their detection classification. We used *Revolver* to extract ASTs from the pages analyzed by Wepawet, and to perform the similarity processing described in the previous sections.

As our processing infrastructure, we used a cluster of four worker machines to process submissions in parallel with the Oracle. Notice that all the steps in *Revolver*'s processing can be easily parallelized. In terms of performance, we managed to process up to 591,543 scripts on a single day, which was the maximum number of scripts that we got on a single day from the Oracle during our experiments.

We will now discuss the parameters that can be tuned in our algorithms (discussed in §4.2), explaining the concrete values we have chosen for our experiments.

Minimum tree size (τ_z). We chose 25 nodes as the minimum size of the AST trees that we will process before combining them to their parent. Smaller ASTs can result from calls to *eval* with tiny arguments, and from calls to short event handlers, such as *onLoad* and *onMouseOver*. We expect that such small ASTs correspond to short scripts that do not implement any interesting functionality alone, but complement the functionality of their parent script.

Minimum pattern size (τ_p). Another threshold that we set is the minimum pattern size. Any node sequence that is repeated more than this threshold is truncated to the threshold value. The primary application of pattern detection is to handle similar packers that decode payloads of different size. We chose 16 for this value, as current packers either work on relatively long arrays (longer than 16, and thus detected) or on single strings (one node, and thus irrelevant to this issue). This amount also excludes the possibility of compressing interesting code sequences, since we rarely see such long patterns outside of packed payloads. Reducing this value would have the effect of making the tree similarity algorithm much more lax.

Nearest neighbor threshold (τ_n). In the nearest neighbors computation, we discard node sequences that are farther than a given distance d from the node sequence currently being inspected. We empirically determined a value for this parameter, by evaluating various

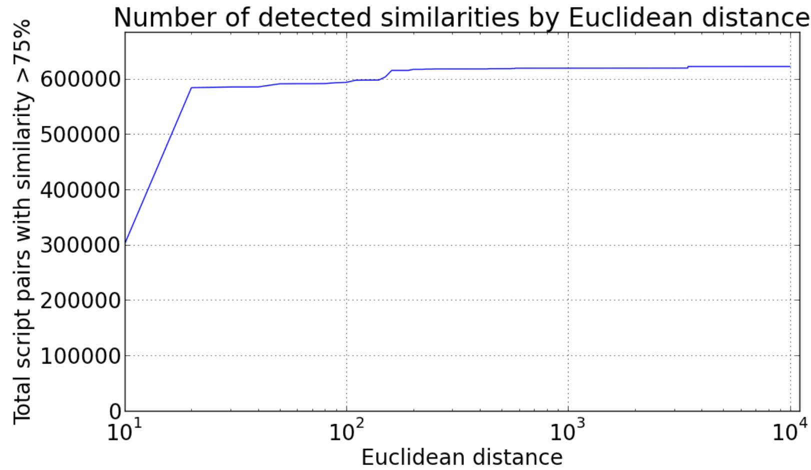

Figure 4.5: Number of detected similarities as a function of the distance threshold.

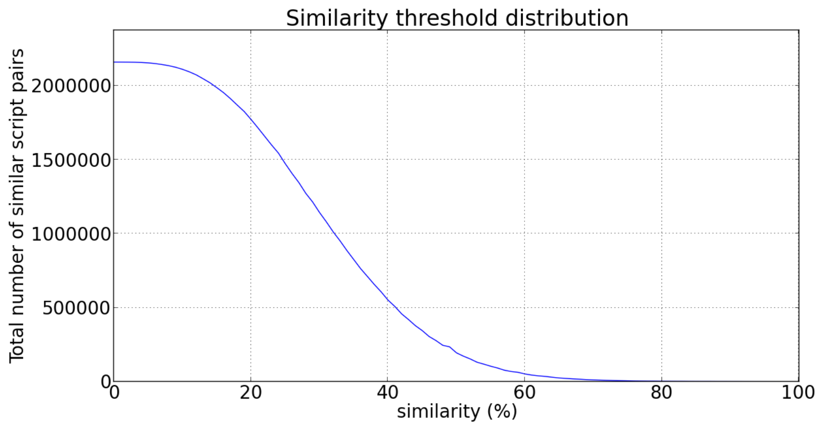

Figure 4.6: The resulting amount of similarities for different similarity thresholds.

values for d and inspecting the resulting similarities. From Figure 4.5, it is apparent that the amount of similarities that are detected levels off fairly sharply past $d = 1,000$. We determined that $10,000$ is a safe threshold that includes a majority of trees while allowing the similarity calculation to be computationally feasible.

Normalized node sequence similarity threshold (τ_s). Care has to be taken when choosing the threshold used to identify similar normalized node sequences. Intuitively, if this value is too low, we risk introducing significant noise into our analysis, which will make *Revolver* consider as similar scripts that in reality are not related to each other. On the contrary, if the value is too high, it will discard interesting similarities. Experimentally (see Figure 4.6), we determined that this occurs for similarity values in the 70%–80% interval. Therefore, we chose 75% as our similarity threshold (in other words, only node sequences that are 75% or more similar are further considered by *Revolver*).

4.4 Evaluation

We evaluated the ability of *Revolver* to aid in detecting evasive changes to malicious scripts in real-world scenarios. While *Revolver* can be leveraged to solve other problems, we feel that automatically identifying evasions is the most important contribution to improving the detection of web-based malware.

4.4.1 Evasions in the wild

Revolver identifies possible evasion attempts by identifying similarities between malicious and benign code. Therefore, *Revolver*'s input is the output of any Oracle that classifies JavaScript code as either malicious or benign. To evaluate *Revolver*, we continuously

Category	Similar Scripts	# Groups by malicious AST
JavaScript Injections	6,996	701
Data-dependencies	101,039	475
Evasions	4,147	155
General evolutions	2,490	273
Total	114,672	1,604

Table 4.2: Benign scripts from Wepawet that have similarities with malicious scripts and their classification from *Revolver*.

submitted to *Revolver* all web pages that Wepawet examined. Since September 2012, we collected $6,468,623$ web pages out of which $265,692$ were malicious. We analyzed $20,732,766$ total benign scripts and $186,032$ total malicious scripts. Out of these scripts, we obtained $705,472$ unique benign ASTs and $5,701$ unique malicious ASTs.

Revolver applied the AST similarity analysis described in Section 4.2, and extracted the pairs of similar ASTs. Table 4.2 summarizes the results of classifying these similarities in the categories considered by *Revolver*. In particular, *Revolver* identified $6,996$ scripts where malicious JavaScript was injected, $101,039$ scripts with data-dependencies, $4,147$ evasive scripts, and $2,490$ scripts as general evolutions. We observe that many of these

scripts can be easily grouped by their similarities with the same malicious script. Therefore, for ease of analysis, we group the pairs by their malicious AST component, and identify 701 JavaScript injections, 475 data-dependencies, 155 evasions, and 273 general evolutions. Our results indicate a high number of malicious scripts that share similarities with benign ones. This is due to the fact that injections and data-dependent malicious scripts naturally share similarities with benign scripts and we are observing many of these attacks in the wild.

To verify the results produced by *Revolver*, we manually analyzed all groups categorized as "evasions". For the rest of the categories we grouped the malicious ASTs into families based on their similarities with each other and examined a few similar pairs from each family. We found the results for the JavaScript injection and data-dependencies categories to be correct. The reason why *Revolver* classified a large number of scripts as data-dependencies is due to the extensive use of a few popular packers, such as the Edwards' packer [26]. For example, the jQuery library was previously officially distributed in a packed state to reduce its size.

Of the 155 evasions groups, we found that only five were not intended evasion attempts. We cannot describe all evasions in detail here, but we provide a brief summary for the most interesting ones in the next section.

The pairs in the "general evolutions" category consisted of cases where *Revolver* identified control flow changes in both the benign and malicious scripts. We manually looked into them and did not find any behavior that could be classified as evasive.

4.4.2 Evasions case studies

The evasions presented here exploit differences in the implementation of Wepawet's JavaScript interpreter and the one used by regular browsers. Notice that these evasions can affect Oracles other than Wepawet; in particular, low-interaction honeyclients, such as the popular jsunpack [41] and PhoneyC [75].

We describe in more detail a subset of the evasions that we found from our experiment on real-world data. In the 22 evasion groups described here, we identified seven distinct evasion techniques, and one programming mistake in a malicious PDF.

We found three cases which leveraged subtle details in the handling of regular expressions and Unicode to cause a failure in a deobfuscation routine when executing in the Oracle (on the contrary, regular browsers would not be affected). In another case, the attackers replaced the JavaScript code used to launch an ActiveX exploit code with equivalent VBScript code. This is done because Internet Explorer can interpret VBScript, while most emulators do not support it. In a different case, the evasive code creates a `div` tag and checks for specific CSS properties, which are set correctly in Internet Explorer but not when executing in our Oracle. We will examine in more detail the next four evasion techniques.

Variable scope inside eval. We found that a successful evasion attack can be launched with minor changes to a malicious script. In one such case, shown in Figure 4.7, the authors of the malicious script changed a *replace* call with a call to `eval`, which, in turn, executed

```
1  // Malicious
2  function foo() {
3  ...
4  W6Kh6V5E4 = W6Kh6V5E4.replace(/\W/g,Bm2v5BSJE);
5  ...
6  }
7  // Evasion
8  function foo(){
9  ...
10 var enryA = mxNEN+F7B07;
11 F7B07 = eval;
12 {}
13 enryA = F7B07('enryA.rep' + 'lace(/\\W/g,CxFHg)');
14 ...
15 }
```

Figure 4.7: Evasion based on differences in the scope handling inside `eval` in different JavaScript engines.

the same *replace*. While this change did not affect the functionality of the script in Internet Explorer, it did change it for our Oracle. In fact, in Wepawet's JavaScript engine, the code inside the `eval` runs in a different scope, and thus, the locally-defined variable on which *replace* is called is not accessible. While the code successfully exploits Internet Explorer, it fails in our Oracle and is marked as benign.

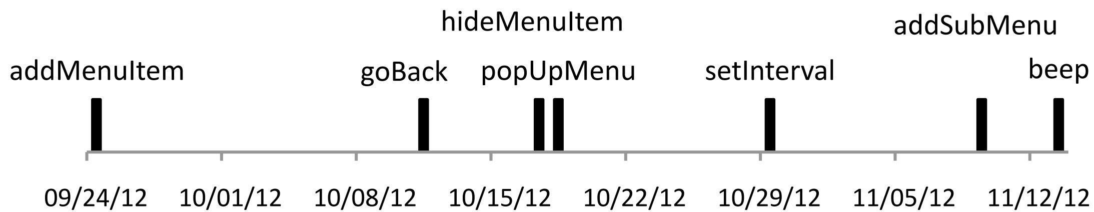

Figure 4.8: Timeline of PDF evasions automatically detected by *Revolver*.

Adobe-specific JavaScript execution. Figure 4.9 shows an evasion that leverages a specific characteristic of the JavaScript engine implementation included in Adobe Reader. In particular, in Adobe's engine (unlike other engines), the `eval` function is accessible as a property of native objects, e.g., a string (line 8) [3]. Wepawet's analyzer is not aware of this difference and fails on scripts that make use of this feature (marking them as benign). The functionally-identical script that does not use this trick, but directly invokes the `eval()` function, is correctly marked as malicious. We also found several instances of evasions related to PDF specific objects, like *app* and *target*, where missing functionality was used to render the malicious script harmless. We show a snippet of this evasion type found in the wild in Figure 4.10.

In Figure 4.8 we see the evasions related to the *app* object that were automatically detected by *Revolver* as found in the wild. Every time *Revolver* detected an evasion there is a spike in the figure, and we made the appropriate patches to Wepawet as soon as possible. What is of particular interest is the fact that attackers react to Wepawet's patching by introducing a new

```
1 // Malicious
2 OlhG='evil_code'
3 wTGB4=eval
4 wTGB4(OlhG)
5
6 // Evasion
7 OlhG='evil_code'
8 wTGB4="this"["eval"]
9 wTGB4(OlhG)
```

Figure 4.9: Evasion based on the ability to access the `eval` function as a property of native objects in Adobe's JavaScript engine.

```
1 if((app.setInterval+/**/"")["indexOf"](aa)!=-1){
2     a=/**/target.creationDate.split('|')[0];}
```

Figure 4.10: Evasion based on PDF specific objects *app* and *target*.

evasion within a few days, making a tool like *Revolver* necessary to automatically keep track of this behavior and keep false negative detections as low as possible.

Evasion through exceptions. Another interesting evasion that *Revolver* found also leverages subtle differences in the JavaScript implementation used in Wepawet and in real browsers. In this case, the malicious script consists of a decryption function and a call to that function. The function first initializes a variable with a XOR key, which will be used

to decrypt a string value (encoding a malicious payload). The decoded payload is then evaluated via `eval`.

The evasion that we found follows the same pattern (see Figure 4.11), but with a few critical changes. In the modified code, the variable containing the XOR key is only initialized the first time that the decryption function runs; in sequential runs, the value of the key is modified in a way that depends on its prior value (Lines 16–17). After the key computation, a global variable is accessed. This variable is not defined the first time the decryption function is called, so that the function exits with an exception (Line 19). On Internet Explorer, this exception is caught, the variable is defined, and the decryption function is called again. The function then runs through the key calculation and then decrypts and executes the encrypted code by calling `eval`.

On our Oracle, a subtle bug (again, in the handling of `eval` calls) in the JavaScript engine caused the function to throw an exception the first *two* times that it was called. When the function is called the third time, it finally succeeds, modifies the XOR key, and attempts to decrypt the string. However, since the key calculation is run *three* times instead of two, the key is incorrect, and the decrypted string results in garbage data. We found three variations of this technique in our experiments.

A very interesting exception-based evasion that we found with *Revolver* was based on the immutability of *window.document.body*. The attacker checks if she can replace the *body* object with a *string*, something that should not be possible and should result in an exception,

but it does not raise an exception in our Oracle because the *body* object is mutable. The interesting part is that we found three completely different malicious scripts evolving to incorporate this evasion, one of them being part of the popular exploit kit *Blackhole 2.0*. This is the first indication that evasion techniques are propagating to different attacking components and indicates that attackers care to keep their attacks as stealthy as possible.

Unicode deobfuscation evasion. This evasion leveraged the fact that Unicode strings in string initializations and regular expressions are treated differently by different JavaScript engines. For example, *Revolver* found two scripts with a similarity of 82.6%. The script flagged as benign contained an additional piece of code that modified the way a function reference to `eval` was computed. More precisely, the benign script computed the reference by performing a regular expression replacement. While this operation executes correctly in Internet Explorer, it causes an error in the JavaScript engine used by Wepawet due to a bug in the implementation of regular expressions.

Incorrect PDF version check. Another similarity that *Revolver* identified involved two scripts contained inside two PDF files, one flagged as benign by Wepawet and the other as malicious. These scripts had a similarity of 99.7%. We determined that the PDF contained an exploit targeting Adobe Reader with versions between 7.1 and 9. The difference found by *Revolver* was caused by an incorrect version check in the exploit code. The benign code mistakenly checked for version greater or equal to 9 instead of less or equal to 9, which combined with the previous checks for the version results in an impossible browser

configuration and as a consequence the exploit was never fired. This case, instead of being an actual evasion, is the result of a mistake performed by the attacker. However, the authors quickly fixed their code and re-submitted it to Wepawet just 13 minutes after the initial, flawed submission.

False positives. The evasion groups contained five false positives. In this context, a false positive means that the similarity identified by *Revolver* is not responsible for the Oracle's misdetection. More precisely, of these false positives, four corresponded to cases where the script execution terminated due to runtime JavaScript errors before the actual exploit was launched. While such behavior could be evasive in nature, we determined that the errors were not caused by any changes in the code, but by other dependencies. These can be due to missing external resources required by the exploit or because of a runtime error. In the remaining case, the control-flow change identified by *Revolver* was not responsible for the misdetection of the script.

Revolver's impact on honeyclients. By continuously running *Revolver* in parallel with a honeyclient, we can improve the honeyclient's accuracy by observing the evolution of malicious JavaScript. The results from such an integration with Wepawet indicate a shift in the attackers' efforts from hardening their obfuscation techniques to finding discrepancies between analysis systems and targeted browsers. Popular exploit kits like *Blackhole* are adopting evasions to avoid detection, which shows that such evasions have emerged as a new problem in the detection of malicious web pages. *Revolver*'s ability to pinpoint, with high

accuracy, these new techniques out of millions of analyzed scripts not only gives a unique view into the attackers' latest steps, but indicates the necessity of such system as part of any honeyclient that analyzes web malware.

4.5 Discussion

As with any detection method, malware authors could find ways to attempt to evade *Revolver*. One possibility consists in splitting the malicious code into small segments, each of which would be interpreted separately through `eval`. *Revolver* is resilient against code fragmentation like this because it combines such scripts back to the parent script that generated them, reconstructing this way the original non-fragmented script.

It is also possible for malware authors to purposefully increase the Euclidean distance between their scripts so that otherwise similar scripts are no longer considered neighbors by the nearest neighbor algorithm. For example, malware authors could swap statements in their code, or inject junk code that has no effect other than decreasing the similarity score. Attackers could also create fully metamorphic scripts, similar to what some binary malware does [56]. We can counteract these attacks by improving the algorithms we use to compute the similarity of scripts. For example, we could use a preprocessing step to normalize a script's code (e.g., removing dead code). A completely different approach would be to leverage *Revolver* to correlate differences in the code of the same web pages when visited by

multiple oracles: if *Revolver* detects significant differences in the code returned during these visits, then we can identify metamorphic web pages. In addition, metamorphic code raises the bar, since an attack needs to be programmatically different every time, and the code must be automatically generated without clearly-detectable patterns. Therefore, this would force attackers to give up their current obfuscation techniques and ability to reuse code.

An attacker could include an evasion and dynamically generate the attack code only if the evasion is successful. The attacker has two options: He can include the evasion code as the first step of the attack, or after initial obfuscation and environment setup. Evasions are hard to find and require significant manual effort by the attackers. Therefore, attackers will not reveal their evasion techniques since they are almost as valuable as the exploits they deliver. Moreover, introducing unobfuscated code compromises the stealthiness of the attack and can yield into detection through signature matching. The second option works in *Revolver*'s favor, since it allows our system to detect similarities in obfuscation and in environmental setup code.

Finally, an operational drawback of *Revolver* is the fact that manual inspection of the similarities that it identifies is currently needed to confirm the results it produces. The number of similarities that were found during our experiments made it possible to perform such manual analysis. In the future, we plan to build tools to support the inspection of similarities and to automatically confirm similarities based on previous analyses.

4.6 Conclusions

In this chapter, we have introduced and demonstrated *Revolver*, a novel approach and tool for detecting malicious JavaScript code similarities on a large scale. *Revolver*'s approach is based on identifying scripts that are similar and taking into account an Oracle's classification of every script. By doing this, *Revolver* can pinpoint scripts that have high similarity but are classified differently (detecting likely evasion attempts) and improve the accuracy of the Oracle.

We performed a large-scale evaluation of *Revolver* by running it in parallel with the popular Wepawet drive-by-detection tool. We identified several cases of evasions that are used in the wild to evade this tool (and, likely, other tools based on similar techniques) and fixed them, improving this way the accuracy of the honeyclient.

```
// Malicious
function deobfuscate(){
    ... // Define var xorkey and compute its value
    for(...) { ... // XOR decryption with xorkey }
    eval(deobfuscated_string);
}
try {
  eval('deobfuscate();')
}
catch (e){
  alert('err');
}

// Evasion
function deobfuscate(){
    try { ... // is variable xorkey defined? }
    catch(e){ xorkey=0; }
    ... // Compute value of xorkey
    VhplKO8 += 0; // throws exception the first time
    for(...) { ... // XOR decryption with xorkey}
    eval(deobfuscated_string);
}
try { eval('deobfuscate();') } // 1st call
catch (e){
  // Variable VhplKO8 is not defined
  try {
    VhplKO8 = 0; // define variable
    eval('deobfuscate();'); // 2nd call
  }
  catch (e){
    alert('ere');
  }
}
```

Chapter 5

Eliciting Malicious Behavior in Browser Extensions

In this chapter we focus on a different threat against the browser: malicious extensions. We present Hulk, a dynamic analysis system that detects malicious behavior in browser extensions by monitoring their execution and corresponding network activity. Hulk elicits malicious behavior in extensions in two ways: *HoneyPages*, which are dynamic pages that adapt to an extension's expectations in web page structure and content, and *fuzzing* so that we can exercise the numerous event handlers that modern extensions heavily rely upon.

All major web browsers today support broad extension ecosystems that allow third parties to install a wide range of modified behavior or additional functionality. Internet Explorer has binary add-ons (Browser Helper Objects), while Firefox, Chrome, Opera, and Safari support JavaScript-based extensions. Some browsers have online web stores to distribute extensions to users. For example, the most popular extension in Chrome's Web Store, AdBlock, has over 10 million users. Other popular extensions serve a variety of functions,

such as preserving privacy, changing the aesthetics of the browser's UI, or integrating with web services such as Google Translate.

The amount of critical and private data that web browsers mediate continues to increase, and naturally this data has become a target for criminals. In addition, the web's advertising ecosystem offers opportunities to profit by manipulating a user's everyday browsing behavior. As a result, malicious browser extensions have become a new threat, as criminals realize the potential to monetize a victim's web browsing session and readily access web-related content and private data.

Our work examines extensions for Google Chrome that are designed with malicious intent—a threat distinct from that posed by attackers exploiting bugs in benign extensions, which has seen prior study [9, 15]. Extensions for Google Chrome are primarily distributed through the Chrome Web Store.[1] Like app stores for other platforms, such as Android or iOS, inherent risks arise when downloading and executing programs from untrusted sources. Reports have documented not only malicious extensions [98], but miscreants *purchasing* extensions (and thereby access to their userbases via update mechanisms) to add malicious functionality [4, 90]. In addition to the web store, extensions can also be directly installed by users and other programs. Installed by a process called *sideloading*, these extensions pose a recognized risk that browser vendors have attempted to prevent through modifications to the browser [62]. Sideloaded extensions are especially problematic since they can be installed

[1] https://chrome.google.com/webstore/category/extensions

without user knowledge, and are not subject to review by a web store. Despite efforts to stifle sideloaded extensions, they remain a significant problem [27].

In this chapter we present Hulk, a tool for detecting malicious behavior in Google Chrome extensions. Hulk relies on dynamic execution of extensions and uses several techniques to trigger malicious functionality during execution. One technique we developed to elicit malicious behavior is the use of *HoneyPages*: specially-crafted web pages designed to satisfy the structural conditions that trigger a given extension. We interpose on all queries and modifications to the DOM tree of the HoneyPage to automatically create elements and mimic DOM tree structures for extensions on the fly. Using this technique, we can readily observe malicious behavior that inserts new `iframe` or `div` elements.

In addition, we built a fuzzer to drive the execution of event handlers registered by extensions. In our experiments, we use the fuzzer to trigger all event handlers associated with web requests, exercising each with 1 million URLs. Although we undertook extensive efforts to trigger malicious behavior, the possibility remains that Hulk lacks the mechanisms to satisfy all of the conditions necessary for eliciting an extension's malicious behavior.

Our analysis of 48,332 Chrome extensions found that malicious extensions pose a serious threat to users. By developing a set of rules that label execution logs from Hulk, we identified 130 malicious extensions and 4,712 "suspicious" extensions, most of which appear in the Chrome Web Store. Several large classes of malicious behavior appear within our set of extensions: affiliate fraud, credential theft, ad injection or replacement, and social network

abuse. In one case, an extension performing ad replacement had nearly 2 million users, similar in size to some of the largest botnets.

In summary, we frame our contributions as follows:

- We present Hulk, a system to perform dynamic analysis for Chrome extensions.
- We demonstrate the effectiveness of HoneyPages and event handler fuzzing to elicit malicious behavior in browser extensions.
- We perform the first broad study of malicious Chrome extensions.
- We characterize several classes of malicious Chrome extensions, some with very large footprints (up to 5.5M installations) and propose solutions to eliminate entire classes of malicious behavior.

5.1 Background

We begin by reviewing the Google Chrome extension model and the opportunities this model provides to malicious extensions.

5.1.1 Chrome Extension Composition

Google Chrome supports extensions written in JavaScript and HTML (distributed as a single `zip` file). A small number of extensions also include binary code plugins, although these are subject to a manual security review process [38]. Each extension contains a (mandatory)

manifest that, along with other extension parameters, describes the permissions the extension uses and the list of resources that the browser should load.

The permission system is designed in the spirit of least privilege, with the goal of limiting the resources available to an extension in case it has exploitable vulnerabilities [9]. The threat model does not attempt to address malicious extensions accessing sensitive content or performing other actions. The permission system determines which sites an extension can access, the allowed API calls, and the use of binary plugins. We describe relevant parts of the permission system later in this section. See Barth et al. for a more detailed description of Chrome's extension architecture [9].

5.1.2 Installing Extensions

The Chrome Web Store is the official means for users to find and install extensions. The Web Store is similar to other app stores, such as those for iOS and Android, in that developers create extensions and upload them to the store for users to download. Extension developers can also push out updates without requiring any action by the end-user.

In addition to the Chrome Web Store, extensions can also be installed manually by a user or an external program. We refer to the installation of extensions outside the web store as *sideloading*. Chrome version 25 (released February, 2013) included changes to prevent silent installation of Chrome extensions and require that the user indicate consent for installation [62]. In May, 2014, Chrome took further steps to prevent sideloading by

requiring *all* installed extensions to be hosted in the Chrome Web Store [50]. While these changes increase the difficulty of sideloading, it is still possible for programs to force silent installation of extensions, since the attacker already has control of the machine. For our study we obtained a set of extensions that are sideloaded into Chrome by other Windows programs, many of which are known malware.

5.1.3 Extension Permissions

Permissions. Chrome requires extensions to list the permissions needed to access the different parts of the extension API. For example, Figure 5.1 shows a portion of a manifest file requesting permission to access the `webRequest` and `cookies` API. The `webRequest` permission allows the extension to "observe and analyze traffic and to intercept, block, or modify requests in-flight" by allowing the extension to register callbacks associated with different parts of the HTTP stack [38]. Similarly, the cookies API allows the extension to get, set, and be notified of changes to cookies.

The extension API permissions operate in conjunction with the optional *host permissions*, which limit the API permissions to access resources only for the specified URLs. For example, in Figure 5.1 the extension requests host permissions for `https://www.google.com/`, which allows it to access `cookies` and `webRequest` APIs for the specified domains. Host permissions also support wildcarding URLs. In Figure 5.1, the extension requests access to `*://*.facebook.com`. This permission allows for access to all subdomains

```
...
"permissions": [
  "cookies",
  "webRequest",
  "*://*.facebook.com/",
  "https://www.google.com/"
],
...
"content_scripts": [
  {
    "matches": ["http://www.yahoo.com/*"],
    "js": ["jquery.js", "myscript.js"]
  }
],
...
"background": {
  "scripts": ["background.js"]
},
...
"content_security_policy": "script-src 'self'
      http://www.foo.com 'unsafe-eval';"
...
```

Figure 5.1: Example of a manifest that shows API permissions for two hosts, followed by content scripts that run on `http://www.yahoo.com`, followed by a background script that runs on all pages. Finally, the CSP specifies the ability to include and `eval` scripts in the extension from `foo.com`.

of `facebook.com` requested via any URL scheme. In addition to wildcards, the special token `<all_urls>` matches any URL.

Besides the permissions described above, we found that extensions request a variety of other permissions. In Section 5.3 we summarize the permissions requested for all of the

extensions we examined, and we discuss the permissions relevant to various types of abuse in Section 5.4. Other resources provide a thorough analysis of the Chrome permission system [9, 15].

Content Scripts. In addition to permissions for accessing various resources associated with a page, extensions can also specify a list of `content_scripts` to indicate JavaScript files that will run inside of the web page. Figure 5.1 shows an example of including two JavaScript files, `jquery.js` and `myscript.js` that will be run in the context of the page for any URLs matching the specified URL patterns (all pages on `http://www.yahoo.com/` in this example). Inside of each JavaScript file the author can include further logic to decide if and when to execute.

The ability to run in the context of a page is a powerful feature. Once a content script executes, any resulting actions become indistinguishable from actions performed by JavaScript provided by the web server. Not only can the scripts modify the DOM tree or other scripts, but they can also issue authenticated web requests (such as POST with proper cookies).

Background Pages. Besides the content scripts that allow an extension to interact with a given page, Chrome also allows extensions to run scripts in a "background page". Figure 5.1 shows an example manifest file that specifies `background.js` as a background page. Background pages often contain the logic and state an extension needs for the entirety of the browser session and do not have any visibility to the user. For example, an extension requesting `webRequest` permissions may use the background script to attach a listener

to read outgoing requests using the `chrome.webRequest.onBeforeRequest.addListener()` call. After filtering on the *host permissions*, Chrome will send the extension a notification for every outgoing request. We detail further examples in the context of the extensions in the following sections.

Content Security Policy. In general, servers can specify a Content Security Policy (CSP) header that the browser uses to determine the sources from which it can include objects on the page. CSP can also specify other options, such as whether to allow the page to perform an `eval` or to embed inline JavaScript [107]. Extensions can use the same syntax to express their CSP in the manifest file. For example, an extension that wishes to include source from `foo.com` and to execute `eval` can specify its CSP as shown in Figure 5.1.

5.2 Architecture

In this section, we describe the architecture of Hulk, our dynamic analysis system that identifies malicious behavior in Chrome extensions. Hulk dynamically loads extensions in a monitored environment and observes the interaction of extensions with the loaded web pages. Using a set of heuristics to identify potentially dangerous behavior, it labels extensions as *malicious*, *suspicious*, or *benign*. In the rest of this section we describe how Hulk works and the challenges that arise in analyzing browser extensions.

5.2.1 Profiling Extensions

At the core of our dynamic analysis system is an instrumented browser and extension loader that enables us to automatically install extensions and instrument activity during web browsing. Our monitoring hooks collect data from multiple vantage points within the system as Hulk visits web pages and triggers a range of extension behavior.

URL Extraction. Before we dynamically analyze an extension we need to ensure that we can trigger the extension's functionality. Most extensions interact with the content of web pages, so we need to choose which URLs to load for our analysis. To this end, we use three sources of URLs: the manifest, the source code, and a list of popular sites. First, using the manifest file of the extension we construct valid URLs that match the permissions and content scripts specified. In some cases, the host permissions of an extension are restrictive—for example, `https://*.facebook.com`—so we can generate URLs that will match the pattern. It is more difficult to pick URLs to visit in cases where the extension requests host permissions on all URLs (Section 5.1.3), because the malicious behavior may only trigger on a small subset of sites. Therefore, we search the source code for any static URLs and visit those as well. Finally, for every extension we also visit a set of popular sites targeted by malicious extensions. We constantly strive to improve this list as we detect malicious extensions attacking particular domains. We however note that although

we use multiple sources of URLs to determine the appropriate pages to visit, our approach is not complete; we discuss the limitations further in Section 5.6.

HoneyPages. Some extensions activate based on the content of a web page instead of the URL. To analyze such extensions we use specially crafted pages that attempt to satisfy the conditions that an extension looks for on a page before performing an action. We call these *HoneyPages*. HoneyPages contain JavaScript functions that overload built-in functions that query the DOM tree of the web page. As a result, when an extension queries for the presence of a specific element we can automatically create it and insert it into the page. For example, if the extension queries an `iframe` DOM element with the intention to alter it, then our HoneyPage will create an `iframe` element, inject it in the DOM tree, and return it to the extension.

HoneyPages enable us to supplement the URL extraction phase and dynamically create an environment for the extension to perform as many actions as it needs. The on-demand nature of a HoneyPage does not restrict us to a specific DOM tree structure, but enables us to determine what an extension looks for in a page during execution, since we can record all interactions within a HoneyPage. By using HoneyPages we can better understand how the extension will behave on arbitrary pages that are otherwise difficult to generate prior to analysis.

5.2.2 Event-Based Execution

The Chrome browser offers to extensions an event-based model to register callbacks that respond to certain browser-level events. For example, extensions use the `chrome.webRequest.onBeforeRequest` callback to intercept all outgoing HTTP requests from the browser. HoneyPages will not trigger callbacks for network events that require special properties, such as a specific URL or HTTP header. Therefore, we complement HoneyPages with event handler fuzzing. Specifically, we invoke all event callbacks that an extension registers in the `chrome.webRequest` API with mock event objects. We point to a HoneyPage loaded in the active tab while invoking the callbacks, enabling us to monitor the changes that the extension attempts to make on that page. Our approach allows us to test for every extension the extension's callbacks on the top 1 million Alexa domains in under 10 seconds on average.

Monitoring Hooks

Browser Extension API. Depending on the permissions included in the manifest (Section 5.1.3), an extension can use the Chrome extension API to perform actions not available to JavaScript running in a web page. As such, monitoring the extension API captures a subset of the total JavaScript activity that results from an extension, but gives us a detailed picture of what the extension attempts to do. For example, we monitor the extension API

and log if the extension registers a callback to intercept all HTTP requests performed by the browser, and then track the changes that the extension makes to the HTTP requests. To do this, we leverage the current logging infrastructure offered by Chrome for monitoring the activity of extensions. We build upon the JavaScript function call logging provided by the browser to identify malicious behavior, such as tampering of security-related HTTP headers.

Content Scripts. We intercept and log all additional code introduced by the extension in the context of the visited page. Doing so provides a more complete picture of the extension's functionality, since it can include remote scripts from arbitrary locations and inject them into the page. Remote scripts can compromise the page's security similar to third-party JavaScript libraries [77], and make the analysis of the extension more difficult. Using remote scripts gives miscreants the ability to blacklist IP addresses of our analysis system (i.e., cloaking [49, 105]) or return code without the malicious components. Remote JavaScript inclusion also renders static analysis on the extension's code fundamentally incomplete since parts of the extension's codebase are not available until execution.

Network Logging. We use a transparent proxy that intercepts all browser HTTP and DNS traffic to log the requests made during extension execution. A browser extension has a set of files available as resources loaded by the browser, and it can also download and execute content from the web. Since the URLs retrieved can be computed at runtime, monitoring the network activity of the extension is critical for a complete analysis of its source code and included components. In addition to identifying remote content, we log all domains

contacted by monitoring the DNS requests generated by the browser. Doing so enables us to identify extensions that contact non-existent domains, which can occur because the extension is no longer operational or up-to-date. In these cases, our analysis was necessarily incomplete, since when the domain was active the extension could have fetched more remote code from it.

5.2.3 Detecting Malicious Behavior

As described in the previous section, our dynamic analysis system can provide detailed information about all browser and extension activity performed while visiting web pages. We combine this data to label the extension as either *benign*, *suspicious*, or *malicious* by applying a set of labeling heuristics based on the behavior. Labeling an extension as malicious indicates we identified behavior harmful to the user. Suspicious indicates the presence of potentially harmful actions or exposing the user to new risks, but without certainty that these represent malicious actions. Finally, when we do not find any suspicious activity, we label the extension as benign.

JavaScript Attributes

We use our monitoring modules described in Section 5.2.2 to identify malicious JavaScript execution. Below we detail actions that we consider malicious or suspicious in our post-processing analysis.

Extension API. As described earlier, Chrome's extension API offers privileged access to additional functionality of the browser besides native JavaScript, using permissions specified in the manifest file. While there are benign uses for every permission, we found several extensions that abuse the API. Specifically, for reasons described below, we consider the following actions available only through the extension API as malicious: uninstalling other extensions, preventing uninstallation of the current extension, and manipulating HTTP headers.

We consider uninstalling other extensions as malicious because some extensions uninstall cleaner extensions, such as the extension Facebook created to remove harmful extensions on its blacklist.[2] We detect this behavior by monitoring the `chrome.management.uninstall` API calls. To avoid false positives, we can differentiate cleaners from malicious extensions because, to the best of our knowledge, cleaners operate in a different fashion than Antivirus does: they clean up malicious extensions and then remove themselves from the browser. This differs from the behavior of malicious extensions, which remain persistent on the system.

Besides attempting to uninstall other extensions, malicious extensions often prevent the user from uninstalling the extension itself. More specifically, we found extensions that prevent the user from opening Chrome's extension configuration page where a user can conveniently uninstall any extension. To prevent uninstallation, malicious extensions interfere

[2] `https://chrome.google.com/webstore/detail/facebook-malicious-extens/mhkafblddkepdhhjpmedkngigkjjknoa`

with tabs that point to the extension configuration page, `chrome://extensions`, either by replacing the URL with a different one, or by removing the tab completely. For analysis, we load a tab with `chrome://extensions` in the browser during our dynamic analysis and monitor any interactions to identify such behavior.

Lastly, using callbacks in the `webRequest` API, a malicious extension can manipulate HTTP headers. Extensions can use the `webRequest` API to effectively perform a man-in-the-middle attack on HTTP requests and responses before they are handled by the browser. This behavior is often malicious (or at least dangerous) since we found extensions that remove security-related headers, such as Content-Security-Policy or X-Frame-Options, through the use of callbacks such as `webRequest.onHeadersReceived` and `webRequestInterval.eventHandled`. By monitoring the use of this API, we can log events that reveal state of HTTP headers before and after the request. Upon manipulation of any security-related headers, we label the extension as malicious.

Interaction with visited pages. In addition to the extension API, we also monitor an extension's use of content scripts to modify web content loaded in the browser. In our analysis, we flag two kinds of interaction: sensitive information theft as malicious and injection of remote JavaScript content as suspicious.

There are many ways an extension can steal personal information from the user. For example, it can act as a JavaScript-based keylogger by intercepting all keystrokes on a page. Extensions can also access form data, such as a password field, before it is encrypted and

sent over the network. Finally, extensions can also steal sensitive information from third parties by accessing sites with which the user has a valid session, and either issuing requests to exfiltrate data, or simply stealing valid authentication tokens.

We label any extension that injects remote JavaScript content into a web page as suspicious. We define this activity as adding a `script` element with a `src` attribute pointing to a domain that is different from the one of the web page. Including these scripts complicates analysis since the JavaScript content can change without any corresponding change in the extension. We have observed changes to JavaScript files that substantially alter the functionality of an extension, possibly due to a server compromise.

Network Level

By monitoring network requests, including DNS lookups and HTTP requests, we identify other types of suspicious/malicious behavior. Using a manual analysis of network logs we have identified two attributes that indicate malicious or suspicious behavior: request errors and modification of HTTP requests. To detect HTTP modifications, we examine if the network response that we observe on the wire differs from the network response finally processed by the browser.

As we discussed earlier, the extension API offers callbacks to give extensions the ability to intercept and manipulate web requests. Not only can extensions drop security-related headers, but extensions can change or add parameters in URLs before the HTTP request is

sent. We find such suspicious behavior common, especially among extensions that request permissions on shopping-related sites such as Amazon, EBay, and others. In these cases, the extension adds parameters to the URL that indicate that the site should credit a particular affiliate for any resulting sales. We discuss this behavior in more detail in Section 5.4. At the network level, we have the complete view of how the requests originally appeared. We combine that knowledge with our `chrome.*` API monitoring to identify the exact changes made to the request.

We also look for errors during domain name resolution to identify extensions that contact domains since taken down. As with drive-by downloads, we expect that malicious code dynamically loaded into an extension will eventually become blacklisted. In such cases, the extension will fail to introduce more code during its execution. We detect this behavior and mark it as suspicious.

5.2.4 Injected Content Analysis

A Chrome extension can also manipulate the visited pages of the browser by injecting a content script. The injected script runs in the context of the visited page and thus has full access to its DOM tree. The injected code can vary significantly, and, with the dynamic nature of JavaScript, can prove difficult to analyze statically. The use of HoneyPages enables us to understand the injected code's full intentions. Instead of trying to infer what the code will do, we actually run it to observe its effects on the DOM tree and classify it accordingly.

Analysis result	**Count**
Malicious	130
Suspicious	4,712
Benign	43,490
Total	48,332

Table 5.1: Classification distribution of extensions.

For example, if the injected code looks for a form field with the name “password,” we classify it as malicious, since it can potentially hijack the user’s credentials on the page. Another example concerns injecting additional code, where the injected code is part of a two-stage process that fetches yet more code from the web and dynamically executes it in the context of the visited page. By relying on HoneyPages to understand the code’s intentions by the effect that the code has on a given page, we obtain a more precise view of what the code attempts do than we can using only static analysis.

5.3 Results

To evaluate Hulk we use two sources of extensions: the official Chrome Web Store (totaling 47,940 extensions), and extensions sideloaded by binaries. We obtained the latter based on

Detection class	Count
[s] Injects dynamic JavaScript	2,672
[s] Produces HTTP 4xx errors	2,322
[s] Evals with input $>$128 chars long	451
[m] Prevents extension uninstall	56
[m] Steals password from form	39
[s] Performs requests to non-existent domain	26
[m] Contains keylogging functionality	23
[m] Injects security-related HTTP header	11
[m] Steals email address from form	10
[m] Uninstalls extensions	8

Table 5.2: Distribution of detected suspicious/malicious behavior from analyzed extensions. Notice that an extension might have more than one detections and that we mark with [m] detections classified as malicious and with [s] detections classified as suspicious.

binaries executed in Anubis [6], which, after removing a large number of duplicates, resulted in a set of 392 unique extensions. As shown in Table 5.1, in total we analyzed 48,332 distinct extensions, of which Hulk labeled 130 as *malicious* and 4,712 as *suspicious*. Table 5.2

Rank	Top 10 types of permissions	# ext.
1	tabs	16,787
2	notifications	12,011
3	unlimitedStorage	9,424
4	storage	5,725
5	contextMenus	4,774
6	cookies	2,872
7	webRequest	2,849
8	webRequestBlocking	2,102
9	webNavigation	1,623
10	management	1,533

Table 5.3: The top 10 permissions found in the manifest files for all extensions we ran. Extensions can include more than one permission.

summarizes all of the detected behaviors, which we analyze in more detail in the following sections.

5.3.1 Permissions Used

In this section we characterize the extensions we executed by identifying the most popular permissions, content scripts, and API calls that they performed.

Permissions. Table 5.3 shows the top 10 permissions from 30,392 unique extensions that use the Chrome Extension API (excluding the host permissions). The most commonly used, the `tabs` permission, allows an extension to interact with the browser's tabs, including navigating a tab to a specified URL and registering callbacks to react to changes in the address bar. The second most popular permission, `notifications`, allows an extension to generate custom notifications that alert the user. The `storage` and `unlimitedStorage` permissions allow storing of permanent data in the user's browser. The `contextMenus` permission allows an extension to add additional items on the context menu of the browser. Context menus appear when the user right clicks on a page. To manipulate the browser's cookies, an extension needs to ask for the `cookies` permission. The permissions `webRequest`, `webRequestBlocking` and `webNavigation` allow an extension to inspect, intercept, block, or modify web requests from the browser. Finally, an extension can get a list of other extensions installed in the browser—and even disable or unistall them—with the `management` permission.

We also computed permission statistics independently for the set of benign extensions and the set of malicious or suspicious ones. To our surprise, we found that permissions for benign

extensions do not differ significantly from permissions requested by malicious/suspicious ones, indicating that often attackers do not need to target different APIs to perform their attacks; maliciousness instead manifests in the way they use the API.

We found 18,313 extensions that use host permissions to restrict on which pages the extension can use the privileged `chrome.*` API. Table 5.4 shows the top 25 hosts appearing in host permissions. As seen in the table, extensions typically request broad permissions using wildcards in URL patterns. In addition these, we examined the hosts that extensions specified as targets for injecting content scripts, per Table 5.5, finding similar broad declarations. In practice, extension authors often use content scripts and host permissions in an unrestricted fashion.

API calls. Table 5.6 shows the top 15 Chrome Extension API calls made during by extensions during our experiments. There are several measurement artifacts introduced by our methodology. To load an extension for testing, we install the extension on a clean browser each time we start an analysis. This causes `runtime.onInstalled` to appear in every analysis independent of the extension's activities. We also open the `chrome://extensions` tab from inside the extension to determine if the extension interferes with the management of extensions. This causes Hulk to record a large number of `tabs.create` calls. In Table 5.6 the `tabs` API is by far the most used API, which matches the popularity of `tabs` permissions observed in Table 5.3.

5.3.2 Network Level

Using network activity alone we identified 24 malicious extensions. These extensions were labeled as malicious by Hulk because they tampered with security-related HTTP headers. By removing HTTP response headers like *Content-Security-Policy*, the malicious extensions can inject JavaScript into pages that specifically do not allow scripts from external sources (according to the CSP policies provided by the web server). For example, Hulk found multiple variants of an active extension on the Chrome Web Store targeting users that seek to cheat in online games; these extensions, generally going by the name "*Cheat in your favorite games*", affect over 20K users.

During our experiments we encountered cases where our analysis could not obtain the full set of information needed to make a decision regarding the maliciousness of an analyzed extension. This problem arose due extensions performing HTTP requests that either returned errors, such as an HTTP 404 responses, or having domain names that no longer resolved. In such cases, given our inability to exercise the extension's full set of capabilities, and because the failed requests might correspond to fetching additional code, we mark these extensions as suspicious.

5.3.3 Extensions Management

Using signals tailored to detect the manipulation of the `chrome://extensions` page (as described in Section 5.2.3), we found several extensions on the Chrome Web Store that prevent uninstallation. Two of of these extensions claim to be video players (each with thousands of user) and completely replace Chrome's extensions management with a page that prevents users from uninstalling them. These are "HD Video Player" with 7,173 users and "SmartScreen Video Plugin" with 11,012 users. These signals also generated a false positive: the "No Tab Left Behind" extension (with only 8 users) allows only one tab at a time to be open. Thus, during our execution this extension prevented us from opening the extension settings tab.

5.3.4 Code Injection

Code injection was the most commonly detected "suspicious" feature in our dataset. In principle injection need not occur at all, since Chrome extensions can come packaged with all the code needed to operate. In total, we found more than 3,000 extensions that dynamically introduced remotely-retrieved code either through script injections or by evoking `eval`. As we noted earlier, using remote code renders static analysis on the extension's code fundamentally incomplete. However, Hulk can identify code injections and pinpoint the remote locations from which an extension fetches code. Although not necessarily malicious,

we found many cases of dangerous code injection. For example, our system identified an extension named "Bang5TaoShopping assistant" from the Chrome Web Store that has been installed in 5.6 million (!) browsers and injects code into every visited page. Several extensions perform this same activity, while others insert tracking pixels for similar purposes. One instance sends cleartext HTTP request to a server controlled by the extension that encodes the URL visited by the user along with a unique identifier, leaking users browsing behavior and thus compromising their privacy.

5.4 Profiting from Maliciousness

In this section, we discuss five categories of malicious behavior in extensions, and describe their characteristics and the methods they employ to carry out their goals. We base each of these categories on examples we found in our feeds. When the extension is available on the Chrome Web Store, we also when possible include the number of users prior to reporting the extension to Google for review.

We have reported to Google any extension that performs behavior that is clearly abusive or malicious, and several of our reports have lead to removals of extensions from the web store.

5.4.1 Ad Manipulation

Advertisement manipulation falls in a grey area in that it does not subvert the user, but rather manipulates an external ecosystem. Replacing ads might appear benign to end users, but removes the potential for monetary credit for website owners (publishers) and instead fraudulently credits the extension owner. We include in this category the addition of new ads as well as the replacement of existing ads or identifiers. We find a range of behaviors in extensions, such as replacing banner ads with different identically-sized banners; inserting banners and text ads into well-known sites (such as Wikipedia); changing affiliate IDs for ads; or simply overlaying ads on top of content. Each instance aims to profit from impressions or clicks on the substituted advertisements.

As one striking example of ad manipulation we found an extension on the Chrome Web Store that had 1.8M users at the time we detected it. The extension, named "SimilarSites Pro" used primarily unobfuscated JS to perform benign functionality as advertised on the Chrome Web Store; however, it also inserted a script element into the content of web pages that downloads another, fully-obfuscated script (using `eval` and `unescape`) from a web server. At the time of analysis, this script contained a large conditional block that looked for `iframe` elements of particular sizes, such as 728x90 pixels, and replaced them with new banners of the same size. Since our first analysis, we have seen several new versions of the script available from the same URL. In addition, the extension contains a blacklist of sites

```
"content_scripts": [{
  "matches": ["http://*/*", "https://*/*"],
  "js": ["js/content.js"]
}],
"permissions": ["http://*/*",
  "https://*/*", "tabs"],
```

Figure 5.2: Permission-related JSON from the manifest file of an extension performing ad replacement.

and meta keywords where it should *not* change the banners, which appears due to many ad networks prohibiting the display of their ads on porn sites.

We find the same JavaScript included in five other extensions from the Chrome Web Store, as well as one sideloaded extension. Based on manual analysis, these extensions are primarily produced by a single company called "SimilarGroup" that engages in dubious behavior through the Chrome Web Store.

To perform banner replacement, the extension requests the permissions shown in Figure 5.2. Such exceptionally wide permissions are not uncommon [15]. Therefore, their presence alone provides little insight into the functionality of the extension. The most significant permission in Figure 5.2 is the broad use of content scripts that allow the extension to inject dynamic JavaScript files from a remote location. Following injection, execution continues as though the page had included it. Such content scripts provide an exceptionally powerful feature to enable a variety of malicious behaviors, as further discussed in this Section.

5.4.2 Affiliate Fraud

Many major merchant web sites such as `amazon.com`, `godaddy.com`, and `ebay.com` run affiliate programs that credit affiliates with a fraction of the sales made as a result of customers referred by the affiliates. Usually merchant programs assign unique identifiers to affiliates, which affiliates then include in the URL that refers customers to the merchant site. Furthermore, affiliate programs usually associate a cookie with the user's browser so that they can attribute a sale to an affiliate within several hours after a user originally visited the merchant site with an affiliate identifier.

As an example, when a user reads product reviews on an Amazon affiliate's blog and clicks on a link to Amazon, the link includes an Amazon affiliate ID specified with the `tag` parameter in the URL, such as `http://www.amazon.com/dp/0961825170/?tag=affiliateID`. When Amazon receives this request, it returns a `Set-Cookie` header with a cookie that associates the user with the affiliate. When the customer returns to Amazon within 24 hours and makes a purchase, Amazon credits the affiliate with a small percentage of the transaction amount.

Such programs expect affiliates to bring potential customers to their sites via affiliate pages that advertise the merchant products. However, we found examples of several extensions involved in *cookie stuffing*—a technique that causes the user's browser to visit the merchant URLs without the user clicking on affiliate URLs. Doing so causes the merchant to deliver

a cookie associated with the fraudulent affiliate, who then receives credit for any future, unrelated purchase made by the customer on the merchant site. Besides defrauding the merchant, the fraudulent affiliate also causes an over-write of the cookie associated with any legitimate affiliate who might have genuinely influenced the user to buy the product.

In our study, we found two kinds of extensions that defrauded affiliate programs. The first group includes extensions that provide some utility to users—such as refreshing pages automatically every few seconds, or changing the theme of popular sites like Facebook—but do not inform users of the extension author profiting from the user's web browsing. Generally, these activities involve monitoring visited URLs for merchant sites where the extension can earn a commission and modifying the outgoing requested URLs to include the affiliate ID, or by injecting `iframe`'s that include affiliate URLs.

For example, we found an extension named "*Split Screen*" (with 52K users) that allows users to show two tabs in a single window, while also stealthily monitoring the URLs visited by the user. It then silently replaces the requested URL with the affiliate's URL for sites such as `amazon.com`, `amazon.co.uk`, `hotelscombing.com`, `hostgator.com`, `godaddy.com`, and `booking.com`. For some merchants, it also sets the referrer header for outgoing requests to falsely imply a visit through the affiliate's site. The extension is able to make these changes using `tab` and `webRequest` permissions, as well as by registering callbacks on `chrome.tabs.onUpdated` to identify changes in the URL as a user types, and `chrome.webRequest.onBeforeSendHeaders`

to modify the referrer header before the browser sends a request to a merchant site. We found four other extensions created by the same developer that similarly provided some small utility to the user while defrauding merchant programs in the background. Overall this developer's extensions have nearly 70K users.

Another extension we found named "Facebook Theme: Basic Minimalist Black Theme" (2.5K users) allows users to change the appearance of Facebook. Besides its stated intent, however, it also monitors browsing and appends an affiliate identifier to 7 different Amazon sites. By using its Content Security Policy (Section 5.1.3) to perform `eval`, it runs a highly-obfuscated hexadecimal and base64-encoded background script that stores all affiliate identifiers in Chrome's storage (using `storage` permissions), and registers callbacks on tab update events using `tab` permissions. When the user visits any URL, Chrome notifies the extension, and the extension uses regular expressions to identify target Amazon URLs for which to add an affiliate identifier. The extension then updates the URL before the browser sends the request. The creator of the extension appears well aware that the extension violates Amazon's Conditions of Use [5] and has heavily used obfuscation, evidently to evade any static analysis for detecting affiliate fraud.

As another example, we found an extension named "Page Refresh" (200 installations) that allows users to refresh tabs periodically and only requests `tabs` permission. By using the background page to listen on all tab update events, if a user visits a merchant site it sets the URL in the tab to a URL shortener that redirects the user to the same merchant page but

with the affiliate identifier included in the URL, thereby stuffing a cookie into the user's browser. This extension abuses 40 different merchants, again including Amazon.

This approach has the advantage that it capitalizes on organic traffic to merchant sites, which can make fraud detection difficult because merchants see visit behavior highly similar to that they would otherwise see as a result of legitimate affiliate referrals.

The second group of extensions includes extensions that clearly state in their descriptions that the extension monetizes the user's online purchases—generally for charitable causes or donations to organizations. The intent or legitimacy of such programs is difficult to ascertain. For example, the extension "Give as you Live" [17] has over 11K users, and forms part of a larger campaign [16] to raise funds for charities from user purchases online. The extension works by adding a list of stores for which the extension author has signed up as an affiliate to the results of major search engines. It also adds a script on merchant sites such as `amazon.co.uk` to redirect users via its own URL. While it does bring legitimate and likely well-intentioned traffic to Amazon, the legitimate affiliates can lose out if users choose to read product reviews on affiliate sites and then make the purchase via this extension.

In fact, a plethora of extensions exists allowing users to donate to charity simply by shopping online. Another such extension uses `webRequest` permissions to modify the requested URL to the affiliate URL, including over-writing the existing affiliate URL. While

this clearly constitutes cookie-stuffing, the extension advertises itself as “Help support our charity by shopping at amazon.co.uk”.[3]

5.4.3 Information Theft

Information theft clearly reflects malicious behavior that has the potential to harm the user in a number of ways, from disclosing private information to financial loss. This broad category of abuse in many ways replicates the functionality of some malware families. Within the browser, we observe stealing of: keypresses, passwords and form data, private in-page content (e.g., bank balances), and authentication tokens such as cookies. We do not include extensions that simply re-use existing authentication tokens already present, such as extensions that spam on social networks; we discuss these in Section 5.4.4.

One example of keylogging we found in the Chrome Web Store, “Chrome Keylogger”, is an experimental extension from researchers [35] that is now removed. Keyloggers use content scripts to register callbacks for key press events, recording the pressed key by using the messaging API to communicate with a background page. The background page then queues up data to send to a remote server. This behavior has similarities with that of extensions that steal form data, although the specific event handlers differ. Both form field theft and keylogging require the extension to specify a content script but do not require other permissions.

[3] The extension creator also helpfully marked the JavaScript code that adds the affiliate identifier as something to obfuscate in the future.

5.4.4 OSN Abuse

Online social network abuse constitutes the final category of prevalent malicious extensions we found. These extensions typically target Facebook, and spread via both the Chrome Web Store and sideloading. These extensions use existing authentication data to interact with the APIs and websites of online social networks. Previous work identified and reported Chrome extensions that abuse social networks, reporting that thousands of users had installed extensions from the Chrome Web Store that spam on Facebook [7].

We found a number of extensions that post spam messages and use other features provided by social networks, such as the ability to upload and comment on photos or query the social graph. When we execute these extensions with Hulk, the HoneyPage features allows the extensions to create elements and insert them into the DOM tree. While we do not typically inspect the visual results of our executions, in one case we observed an extension creating `div` elements to mimic Facebook status updates and inserting them into a page. The HoneyPage acted as a sink for the spam status messages resulting in a page full of spam for the infected user.

One extension of interest, *WhasApp* (a name closely resembling the popular *WhatsApp*, a mobile chat application), has since been removed from the Chrome Web Store, but we also found evidence of the same extension being sideloaded from malware. The extension targets both Facebook and Tumblr. At Facebook, the extension uploads images to Facebook and

```
"content_scripts": [{
  "js":["BlobBuilder.js", ... ],
  "matches":["http://*/*", "https://*/*" ],
  "run_at": "document_end"
}],
"permissions":["http://*/*", "https://*/*",
"*://*.facebook.com/",
"tabs", "cookies", "notifications",
"contextMenus", "webRequest", ...],
```

Figure 5.3: Permissions and content script excerpts from the manifest for an extension that spams on Facebook and creates Tumblr accounts.

then comments on them with messages containing URLs. In some cases the links are used to spread the malicious extension to a wider audience, while other URLs sought to monetize users as part of a spam campaign to advertise products. At Tumblr, the extension creates new Tumblr accounts and verifies them in the background.

The manifest file contains permissions and content scripts that request broad access, as shown in Figure 5.3. The extension is in fact over-privileged, since the extension in fact does not use some of the API permissions the manifest includes. Prior work has identified over-privileging as not uncommon, even among benign extensions [28]. Figure 5.3 shows the extension specifically requesting access for permissions and content scripts on `facebook.com` in addition to all other sites, which provides a hint as to the sites targeted. To carry out spamming on Facebook and Tumblr account creation, the extension actually only requires the use of content scripts. The abusive component of the extension is 15 lines

of JavaScript that downloads a much larger remote JavaScript file containing the spamming functionality.

5.5 Recommendations

In this section, we frame changes to make Chrome's extension ecosystem safer. Extensions should not have the ability to manipulate browser configuration pages, such as `chrome://extensions`, that govern how users manage and uninstall extensions. Extensions should also not be allowed to uninstall other extensions unless they are from the same author or a trusted source (such as Google or Antivirus vendors). We also recommend preventing extensions from manipulating HTTP requests by **removing security-related headers** that compromise the security of web pages. This change will require modifications to several extension APIs to comprehensively address this issue, the primary one being `webRequest`.

To address cloaking and other changes in remotely included content, we suggest that Google should encourage **local inclusion of static files** in the context of a web page. Chrome supports pushing automatic updates of extensions to users, so remotely including additional JavaScript code is not necessary to support rapid changes in an extenion's code. This change will make it possible to have a more complete analysis of extension behavior,

since the analysis engine—Hulk or otherwise[4]—will have the complete extension code available. To encourage developers to write completely self-contained extensions and not load additional code from the network, one could introduce a new policies, such as: if an extension loads code from a remote site, it loses permissions such as the ability to inject that new code into the visited pages.

Finally, extensions should not have the ability to **hook all keyboard events** on a given site. The *window.onkey** API that exists in JavaScript has utility for pages that want to intercept the keyboard events of their users, but in the context of extensions it provides too much power. An experimental API (*chrome.commands*) exists that allows extensions to register keyboard shortcuts; this strikes us as a step in the right direction, as this covers the common use-case for requiring access to these events.

These suggestions will not eliminate malicious extensions, but can prevent classes of attacks, and significantly facilitate the analysis of extensions.

5.6 Limitations

Our system uses dynamic analysis for analyzing extensions, and, as with every dynamic analysis system, the correct classification of an extension relies on triggering the malicious activity. Hulk employs HoneyPages and event handler fuzzing on the extension's web

[4] In particular, ultimately an extension store operator such as Google needs to undertake such analysis as part of its curation of the store contents.

request listeners to enhance dynamic analysis, but does not provide a complete view of extension behavior. For example, we do not attempt to address cloaking that loads different code based on the client's location or time. We also will not observe behavior that depends on specific targets, such as those that require user interaction with a visited page to take effect. Similarly, pages that require sign-in pose difficulties. Hulk has a pre-set list of sites and credentials to use while visiting pages, but does not perform account creation on the fly.

Hulk's Honeypages do not currently support multi-step querying of DOM elements. While we can place elements in the DOM tree that an extension looks for, if the extension expects elements to have additional properties in order to trigger its malicious behavior, we will fail to adapt to the extension's expectations. We plan on improving HoneyPages to support multi-step querying, and for many element types and attributes this appears possible.

We currently also lack data flow analysis in the Chrome browser, a feature that would substantially improve the depth of behavior available for analysis. One example where this would prove particular useful regards keystroke interception. Without data flow tracking, we cannot automatically derive whether this information ultimately becomes transmitted to a third party via a network request.

Another difficult concern for Hulk is analysis evasion by extensions that specifically look for HoneyPages. A determined adversary with knowledge of the system could try to evade Hulk by querying for random elements in the DOM tree first, and, if found, avoid malicious activity. A similar type of evasive behavior arose for in submissions to Wepawet [49]. One

way to counter this is by introducing non-deterministic HoneyPages for which DOM tree queries only succeed with a given probability. We could further enhance this approach by crawling a few million sites and building models of the existing elements to assign apt probabilities weights for different queries. This approach may also require analysis of an extension's DOM queries in case the extension repeatedly performs these in an effort to detect randomized queries. Finally, we can consider measuring code coverage to examine the impact that each DOM query has on the amount of code executed by an extension, as the extension will skip executing the malicious code when it detects the presence of an analysis system.

5.7 Conclusions

In this chapter we presented Hulk, a system to dynamically analyze Chrome browser extensions and identify malicious behavior. Our system monitors an extension's actions and creates a dynamic environment that adapts to an extension's needs in order to trigger the intended behavior of extensions, classifying the extension as malicious or benign accordingly. In total, we identified 130 malicious and 4,712 suspicious extensions that have up to 5.5 million browser installations, many of which remain live in the Chrome Web Store. Based on these results, we developed a detailed characterization of the malicious behavior that we found, targeted at determining the motivation behind the extension. Finally, we

propose several changes for the Chrome browser ecosystem that could eliminate classes of extension-based attacks and aid with analysis.

Rank	Top 25 hosts in permissions	# ext.
1	http://*/*	7,319
2	https://*/*	6,395
3	<all_urls >	2,044
4	http://*/	1,126
5	*://*/*	1,025
6	https://*/	665
7	www.flashgame90.com/Default.aspx	224
8	https://api.twitter.com/	200
9	http://localhost/*	161
10	http://127.0.0.1/*	133
11	https://secure.flickr.com/	95
12	*://*.facebook.com/*	91
13	*://*/	89
14	https://www.facebook.com/*	82
15	http://vk.com/*	77
16	http://*.facebook.com/*	77
17	https://mail.google.com/*	71
18	https://*.facebook.com/*	70
19	http://*.google.com/	68
20	https://www.google-analytics.com/	67
21	https://mail.google.com/	64
22	https://*.google.com/	62
23	https://twitter.com/*	61
24	https://www.googleapis.com/	60
25	google.com/accounts/OAuthGetAcc[..]	56

Table 5.4: The top 25 host permissions used by extensions. Extensions can include more than one host permission per manifest.

Rank	Top 25 hosts in content_scripts	# ext.
1	http://*/*	12,472
2	https://*/*	10,864
3	<all_urls>	4,795
4	*://*/*	1,536
5	https://www.facebook.com/*	520
6	*://*.facebook.com/*	510
7	https://mail.google.com/*	458
8	http://www.facebook.com/*	433
9	https://*.facebook.com/*	344
10	http://*.facebook.com/*	320
11	file://*/*	315
12	https://twitter.com/*	303
13	http://mail.google.com/*	273
14	*://pages.brandthunder.com/[..]	265
15	https://plus.google.com/*	261
16	ftp://*/*	246
17	http://vk.com/*	227
18	http://www.youtube.com/*	211
19	file:///*	207
20	*://mail.google.com/*	189
21	http://twitter.com/*	179
22	*://www.facebook.com/*	178
23	http://ak.imgfarm.com/images[..]	177
24	*://*.reddit.com/*	164
25	https://vk.com/*	164

Table 5.5: The top 25 hosts used in extensions' content script permissions.

Rank	Top 15 chrome.* APIs called	# calls
1	runtime.onInstalled	182,476
2	webRequestInternal.eventHandled	57,466
3	tabs.getAllInWindow	49,312
4	tabs.onUpdated	32,354
5	tabs.create	25,947
6	i18n.getMessage	13,549
7	webRequest.onBeforeSendHeaders	13,213
8	runtime.connect	13,004
9	extension.getURL	11,942
10	storage.get	10,178
11	contextMenus.create	7,816
12	tabs.get	6,970
13	webRequest.onBeforeRequest	6,168
14	runtime.sendMessage	5,847
15	extension.sendRequest	5,454

Table 5.6: The top 15 chrome.* APIs called by extensions during dynamic analysis.

Chapter 6

Conclusions and Future Work

In this dissertation, I have analyzed continuously-changing web threats that target the browser and have shown several novel techniques to defend against them. I have described how the attackers can use sophisticated ways to evade the state-of-the-art detection systems that exist. To deal with such evasive attempts, I have developed a system that tracks the evolution of attacks, leading to better detection of such threats. I have also showed how we can deal with emerging threats against the browser, such as malicious browser extensions, by building a system that dynamically analyzes them and pinpoints their malicious behavior. The extensions identified as malicious by our system were reported to Google and removed from Chrome's Web Store, affecting millions of users that had them installed.

In the future, I plan to continue working on improving the browser's security and making the Internet a safer place. As attacks become more sophisticated and more targeted, we will need to devise new techniques to eliminate them. I plan to change the way we deal with browser and Internet attacks by introducing a novel analysis model that is integrated into the

browser. This way we will eliminate evasions completely, as the user's system is the one that will analyze the attacking code. By moving the analysis inside the browser we will be able to detect targeted attacks, which was previously not possible by older analysis systems, as only a tiny fraction of web clients observe targeted attacks. In a nutshell, my ultimate goal is to protect all Internet users so that everybody can safely browse the web.

Bibliography

[1] IE 6 Countdown. https://www.modern.ie/en-us/ie6countdown.

[2] Internet Explorer 6. https://en.wikipedia.org/wiki/Internet_Explorer_6.

[3] JavaScript for Acrobat API Reference. http://wwwimages.adobe.com/www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_api_reference.pdf.

[4] R. Amadeo. Adware vendors buy Chrome Extensions to send ad- and malware-filled updates. http://arstechnica.com/security/2014/01/malware-vendors-buy-chrome-extensions-to-send-adware-filled-updates/, Jan 2014.

[5] Amazon. Associates Program Operating Agreement. https://affiliate-program.amazon.com/gp/associates/agreement/, 2012.

[6] Anubis: Analyzing Unknown Binaries. http://anubis.seclab.tuwien.ac.at.

[7] F. Assolini. Think twice before installing Chrome extensions. http://www.securelist.com/en/blog/208193414/Think_twice_before_installing_Chrome_extensions, Mar 2012.

[8] D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient Detection of Split Personalities in Malware. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2010.

[9] A. Barth, A. P. Felt, P. Saxena, and A. Boodman. Protecting Browsers from Extension Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2010.

[10] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, Behavior-Based Malware Clustering. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2009.

[11] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A Tool for Analyzing Malware. In *Proceedings of the European Institute for Computer Antivirus Research Annual Conference (EICAR)*, 2006.

[12] R. Boscovich et al. Microsoft Security Intelligence Report. Technical Report Volume 7, Microsoft, Inc., 2009.

[13] M. Broersma. Web attacks slip under the radar. `http://news.techworld.com/security/10620/web-attacks-slip-under-the-radar/`, 2007.

[14] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *Proceedings of the International World Wide Web Conference (WWW)*, 2011.

[15] N. Carlini, A. P. Felt, and D. Wagner. An Evaluation of the Google Chrome Extension Security Architecture. In *Proceedings of the USENIX Security Symposium*, 2012.

[16] Charles Arthur. Infographic: Internet shopping. `http://www.theguardian.com/technology/blog/2011/jul/04/internet-shopping-infographic-give-as-you-live-charity`, 2011.

[17] Chrome Web Store. Give as you Live. `https://chrome.google.com/webstore/detail/give-as-you-live/fceblikkhnkbdimejiaapjnijnfegnii`, 2013.

[18] M. Cova, C. Kruegel, and G. Vigna. Detection and Analysis of Drive-by-Download Attacks and Malicious JavaScript Code. In *Proceedings of the International World Wide Web Conference (WWW)*, 2010.

[19] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. Zozzle: Low-overhead Mostly Static JavaScript Malware Detection. In *Proceedings of the USENIX Security Symposium*, 2011.

[20] CVE. Windows ANI LoadAniIcon() Chunk Size Stack Overflow (HTTP). `http://cve.mitre.org/cgi-bin/cvename.cgi?name=2007-0038`.

[21] CWSandbox. `http://www.cwsandbox.org/`, 2009.

[22] M. Dhawan and V. Ganapathy. Analyzing Information Flow in JavaScript-Based Browser Extensions. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[23] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[24] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: Malware Analysis via Hardware Virtualization Extensions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2008.

[25] V. Djeric and A. Goel. Securing script-based extensibility in web browsers. In *Proceedings of the USENIX Security Symposium*, 2010.

[26] D. Edwards. Dean Edwards Packer. `http://bit.ly/TWQ46b`.

[27] F-Secure. Coremex innovates search engine hijacking. `http://www.f-secure.com/weblog/archives/00002689.html`, April 2014.

[28] A. P. Felt, K. Greenwood, and D. Wagner. The Effectiveness of Application Permissions. In *Proceedings of the USENIX Conference on Web Application Development (WebApps)*, 2011.

[29] P. Ferrie. Attacks on Virtual Machines. In *Proceedings of the Association of Anti-Virus Asia Researchers Conference*, 2007.

[30] S. Fewer. Reflective DLL injection. `http://www.harmonysecurity.com/files/HS-P005_ReflectiveDllInjection.pdf`.

[31] P. Fogla and W. Lee. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2006.

[32] S. Frei, T. Dübendorfer, G. Ollman, and M. May. Understanding the Web browser threat: Examination of vulnerable online Web browser populations and the "insecurity iceberg". In *Proceedings of DefCon 16*, 2008.

[33] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is Not Transparency: VMM Detection Myths and Realities. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, 2007.

[34] Gargoyle Software Inc. HtmlUnit. `http://htmlunit.sourceforge.net/`.

[35] C. Giuffrida, S. Ortolani, and B. Crispo. Memoirs of a browser: A cross-browser detection model for privacy-breaching extensions. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*. ACM, 2012.

[36] Google. Chromium Development Calendar and Release Info. `https://www.chromium.org/developers/calendar`.

[37] Google. Safe Browsing API. `http://code.google.com/apis/safebrowsing/`.

[38] Google. What are extensions? `https://developer.chrome.com/extensions/index`, 2014.

[39] C. Grier, L. Ballard, J. Caballero, N. Chachra, C. J. Dietrich, K. Levchenko, P. Mavrommatis, D. McCoy, A. Nappa, A. Pitsillidis, N. Provos, M. Z. Rafique, M. A. Rajab, C. Rossow, K. Thomas, V. Paxson, S. Savage, and G. M. Voelker. Manufacturing Compromise: The Emergence of Exploit-as-a-Service. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[40] A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified Security for Browser Extensions. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 115–130. IEEE, 2011.

[41] B. Hartstein. jsunpack – a generic JavaScript unpacker. `http://jsunpack.jeek.org/dec/go`.

[42] T. Holz. AV Tracker. `http://honeyblog.org/archives/37-AV-Tracker.html`, 2009.

[43] T. Jaeger. Reference Monitor Concept. In *Encyclopedia of Cryptography and Security*, 2010.

[44] J. Jang, D. Brumley, and S. Venkataraman. BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.

[45] X. Jiang, X. Wang, and D. Xu. Stealthy Malware Detection and Monitoring through VMM-Based "Out-of-the-Box" Semantic View Reconstruction. *ACM Transactions on Information and System Security (TISSEC)*, 13(2), Feb. 2010.

[46] Joebox: A Secure Sandbox Application for Windows. `http://www.joebox.org/`, 2009.

[47] M. G. Kang, H. Yin, S. Hanna, S. McCamant, and D. Song. Emulating Emulation-Resistant Malware. In *Proceedings of the Workshop on Virtual Machine Security (VMSec)*, 2009.

[48] A. Kapravelos, M. Cova, C. Kruegel, and G. Vigna. Escape from Monkey Island: Evading High-Interaction Honeyclients. In *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2011.

[49] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna. Revolver: An Automated Approach to the Detection of Evasive Web-based Malware. In *Proceedings of the USENIX Security Symposium*, 2013.

[50] E. Kay. Protecting Chrome users from malicious extensions. `http://chrome.blogspot.com/2014/05/protecting-chrome-users-from-malicious.html`, May 2014.

[51] T. Klein. ScoopyNG - The VMware detection tool. `http://www.trapkit.de/research/vmm/scoopyng/index.html`.

[52] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-Cloaking Internet Malware. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2012.

[53] B. Krebs. Former anti-virus researcher turns tables on industry. `http://voices.washingtonpost.com/securityfix/2009/10/former_anti-virus_researcher_t.html`, October 27 2009.

[54] B. Krebs. Virus Scanners for Virus Authors, Part II. `http://krebsonsecurity.com/2010/04/virus-scanners-for-virus-authors-part-ii/`, 2010.

[55] C. Kreibich, N. Weaver, C. Kanich, W. Cui, and V. Paxson. GQ: Practical containment for measuring modern malware systems. In *Proceedings of the ACM Internet Measurement Conference (IMC)*, pages 397–412. ACM, 2011.

[56] F. Leder, B. Steinbock, and P. Martini. Classification and detection of metamorphic malware using value set analysis. In *Proceedings of the Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[57] Z. Li, X. Wang, and J. Y. Choi. Spyshield: Preserving privacy from spy add-ons. In *Proceedings of the Recent Advances in Intrusion Detection (RAID)*, 2007.

[58] M. Lindorfer, C. Kolbitsch, and P. M. Comparetti. Detecting Environment-Sensitive Malware. In *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*.

[59] T. Liston and E. Skoudis. On the Cutting Edge: Thwarting Virtual Machine Detection. `http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf`, 2006.

[60] L. Liu, X. Zhang, G. Yan, and S. Chen. Chrome Extensions: Threat Analysis and Countermeasures. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2012.

[61] L. Lu, V. Yegneswaran, P. Porras, and W. Lee. BLADE: An Attack-Agnostic Approach for Preventing Drive-By Malware Infections. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2010.

[62] P. Ludwig. No more silent extension installs. `http://blog.chromium.org/2012/12/no-more-silent-extension-installs.html`, Dec 2012.

[63] Malwarebytes. Exploit Kits: A Fast Growing Threat. `https://blog.malwarebytes.org/exploits-2/2015/01/exploit-kits-a-fast-growing-threat/`.

[64] Malwarebytes. Fileless Infections from Exploit Kit: An Overview. `https://blog.malwarebytes.org/exploits-2/2014/09/fileless-infections-from-exploit-kit-an-overview/`.

[65] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[66] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi. Testing CPU Emulators. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2009.

[67] Microsoft. What is SmartScreen Filter? `http://www.microsoft.com/security/filters/smartscreen.aspx`.

[68] Microsoft. Microsoft Security Intelligence Report, Volume 13. Technical report, Microsoft Corporation, 2012.

[69] MITRE. HoneyClient. `http://www.honeyclient.org/`.

[70] A. Moshchuk, T. Bragin, D. Deville, S. Gribble, and H. Levy. SpyProxy: Execution-based Detection of Malicious Web Content. In *Proceedings of the USENIX Security Symposium*, 2007.

[71] A. Moshchuk, T. Bragin, S. Gribble, and H. Levy. A Crawler-based Study of Spyware in the Web. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.

[72] Moss. Moss with obfuscated scripts. `http://goo.gl/XzJ7M`.

[73] M. Muja and D. G. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *Proceedings of the Conference on Computer Vision Theory and Applications (VISAPP)*, 2009.

[74] T. Müller, B. Mack, and M. Arziman. Web Exploit Finder. `http://www.xnos.org/security/web-exploit-finder.html`.

[75] J. Nazario. PhoneyC: A Virtual Client Honeypot. In *Proceedings of the USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, 2009.

[76] A. Nguyen, N. Schear, H. Jung, A. Godiyal, S. King, and H. Nguyen. MAVMM: Lightweight and Purpose Built VMM for Malware Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2009.

[77] N. Nikiforakis, L. Invernizzi, A. Kapravelos, S. Van Acker, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. You are what you include: Large-scale evaluation of remote JavaScript inclusions. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2012.

[78] Norman Sandbox. `http://www.norman.com/about_norman/technology/norman_sandbox/`, 2009.

[79] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A Fistful of Red-Pills: How to Automatically Generate Procedures to Detect CPU Emulators. In *Proceedings of the USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[80] J. Pate, R. Tairas, and N. Kraft. Clone Evolution: a Systematic Review. *Journal of Software Maintenance and Evolution: Research and Practice*, 2011.

[81] N. Provos, P. Mavrommatis, M. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the USENIX Security Symposium*, 2008.

[82] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, and N. Modadugu. The Ghost in the Browser: Analysis of Web-based Malware. In *Proceedings of the USENIX Workshop on Hot Topics in Understanding Botnet*, 2007.

[83] T. Ptacek and T. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., 1998.

[84] D. Quist, V. Smith, and O. Computing. Detecting the Presence of Virtual Machines Using the Local Data Table. http://www.offensivecomputing.net/files/active/0/vm.pdf.

[85] T. Raffetseder, C. Kruegel, and E. Kirda. Detecting System Emulators. In *Proceedings of the Information Security Conference*, 2007.

[86] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in Circumventing Web-Malware Detection. Technical report, Google, 2011.

[87] M. A. Rajab, L. Ballard, N. Lutz, P. Mavrommatis, and N. Provos. CAMP: Content-Agnostic Malware Protection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[88] P. Ratanaworabhan, B. Livshits, and B. Zorn. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. In *Proceedings of the USENIX Security Symposium*, 2009.

[89] J. W. Ratclif. Pattern Matching: the Gestalt Approach. *Dr. Dobb's*, 1988.

[90] Reddit. Reddit: I am One of the Developers of a Popular Chrome Extension.... http://www.reddit.com/r/IAmA/comments/1vjj51/i_am_one_of_the_developers_of_a_popular_chrome/, Jan 2014.

[91] K. Rieck, T. Krueger, and A. Dewald. Cujo: Efficient Detection and Prevention of Drive-by-Download Attacks. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2010.

[92] J. Rocaspana. SHELIA: A Client HoneyPot For Client-Side Attack Detection. http://www.cs.vu.nl/~herbertb/misc/shelia/, 2009.

[93] C. K. Roy and J. R. Cordy. A Survey on Software Clone Detection Research. Technical report, School of Computing, Queen's University, 2007.

[94] J. Rutkowska. Red Pill... or how to detect VMM using (almost) one CPU instruction. http://www.invisiblethings.org/papers/redpill.html, 2004.

[95] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A Symbolic Execution Framework for JavaScript. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[96] S. Schleimer, D. Wilkerson, and A. Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, 2003.

[97] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure In-VM Monitoring Using Hardware Virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2009.

[98] M. Ter Louw, J. S. Lim, and V. Venkatakrishnan. Enhancing Web Browser Security Against Malware Extensions. *Journal in Computer Virology*, 4(3):179–195, 2008.

[99] The Honeynet Project. Capture-HPC. `https://projects.honeynet.org/capture-hpc`.

[100] ThreatExpert. `http://www.threatexpert.com/`, 2009.

[101] W. Tsaur, Y. Chen, and B. Tsai. A New Windows Driver-Hidden Rootkit Based on Direct Kernel Object Manipulation. In *Proceedings of the Algorithms and Architectures for Parallel Processing Conference*, 2009.

[102] M. Van Gundy, H. Chen, Z. Su, and G. Vigna. Feature Omission Vulnerabilities: Thwarting Signature Generation for Polymorphic Worms. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[103] A. Vasudevan and R. Yerraballi. Cobra: Fine-grained Malware Analysis using Stealth Localized Executions. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2006.

[104] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Signatures Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2004.

[105] D. Wang, S. Savage, and G. M. Voelker. Cloak and Dagger: Dynamics of Web Search Cloaking. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 477–490. ACM, 2011.

[106] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2006.

[107] M. West. An Introduction to Content Security Policy. `http://www.html5rocks.com/en/tutorials/security/content-security-policy/`, 2012.

[108] H. Yin, P. Poosankam, S. Hanna, and D. Song. HookScout: Proactive Binary-Centric Hook Detection. *Proceedings of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2010.