UNDERSTANDING AND DETECTING NEWLY EMERGING ATTACK VECTORS IN CYBERCRIMES

by

Daiping Liu

A dissertation defense submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical & Computer Engineering

2018

© 2018 Daiping Liu
All Rights Reserved

UNDERSTANDING AND DETECTING NEWLY EMERGING ATTACK VECTORS IN CYBERCRIMES

by

Daiping Liu

Approved: ______________________________
Kenneth E. Barner, Ph.D.
Chair of the Department of Electrical Engineering

Approved: ______________________________
Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: ______________________________
Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation defense and that in my opinion it meets the academic and professional standard required by the University as a dissertation defense for the degree of Doctor of Philosophy.

Signed: ______________________________

Haining Wang, Ph.D.
Professor in charge of dissertation defense

I certify that I have read this dissertation defense and that in my opinion it meets the academic and professional standard required by the University as a dissertation defense for the degree of Doctor of Philosophy.

Signed: ______________________________

Chase Cotton, Ph.D.
Member of dissertation defense committee

I certify that I have read this dissertation defense and that in my opinion it meets the academic and professional standard required by the University as a dissertation defense for the degree of Doctor of Philosophy.

Signed: ______________________________

Xiaoming Li, Ph.D.
Member of dissertation defense committee

I certify that I have read this dissertation defense and that in my opinion it meets the academic and professional standard required by the University as a dissertation defense for the degree of Doctor of Philosophy.

Signed: ______________________________

Kun Sun, Ph.D.
Member of dissertation defense committee

ACKNOWLEDGEMENTS

This dissertation is written with the support and help from many individuals. I would like to thank all of them.

First and foremost, I would like to express my deepest appreciation to my advisor, Dr. Haining Wang. Without his guidance in my research, encouragement in my life, and confidence in my abilities, this dissertation would not have been possible.

I would also like to thank my dissertation committee, Dr. Chase Cotton, Dr. Xiaoming Li, and Dr. Kun Sun, for serving on my Ph.D committee as well as their insightful comments.

My sincere thanks also go to all members of our group past and present, Dr. Shuai Hao, Dr. Haitao Xu, Dr. Jidong Xiao, Dr. Dacuan Liu, Xing Gao, Yubao Zhang, for the stimulating discussions, constructive suggestions, generous assistance, and effective teamwork.

Futhurmore, I would like to thank the faculty and staff at the Department of Electrical & Computer Engineering, University of Delaware. Special thanks to Gwen Looby and Amber Spivey for their considerate and effective assistance.

Last but not the least, I would like to thank my family. Thanks to my parents, whose unwavering love and support has made me who I am today. Thanks to my wife, Ge Peng, for lighting up my life with so much love and joy.

This dissertation was supported in part by the U.S. NSF grant CNS-1618117, as well as ONR grants N00014-13-1-0088 and N00014-17-1-2485.

TABLE OF CONTENTS

LIST OF TABLES . . . . **x**
LIST OF FIGURES . . . . **xi**
ABSTRACT . . . . **xii**

Chapter

1 INTRODUCTION **1**

1.1 Problem Statements 1
1.2 Contributions 3
1.3 Dissertation Organization 5

2 RELATED WORK **6**

2.1 Malicious Document Detection 6
2.2 DNS Security 8
2.3 Malicious Domain Detection 9
2.4 Memory Safety in Software 11

3 DETECTING MALICIOUS JAVASCRIPT IN PDF THROUGH DOCUMENT INSTRUMENTATION **13**

3.1 Introduction 13
3.2 System Design 16

3.2.1 Architecture 16
3.2.2 Static Features 17
3.2.3 Document Instrumentation 19
3.2.4 Runtime Features 22
3.2.5 Runtime Detection and Confinement 25

3.2.6 De-instrumentation 28

3.3 Security Analysis 28

3.3.1 Threat Model 29
3.3.2 Potential Advanced Attacks and Countermeasures 29

3.4 Evaluation 31

3.4.1 Data Collection 32
3.4.2 Feature Validation 32
3.4.3 Detection Accuracy 36
3.4.4 System Performance 39

3.5 Conclusion 42

4 TOWARDS AUTOMATED DETECTION OF SHADOWED DOMAINS 43

4.1 Introduction 43
4.2 Background 46

4.2.1 Basics of Domain Name 46
4.2.2 Domain Shadowing 47
4.2.3 Real-world Example 50

4.3 Automatic Detection of Shadowed Domains 51

4.3.1 Overview 51
4.3.2 Dataset 54
4.3.3 Features of Domain Shadowing 56

4.3.3.1 Subdomain Usage 58
4.3.3.2 Subdomain Hosting 60
4.3.3.3 Subdomain Activity 61
4.3.3.4 Subdomain Name 63

4.4 Evaluation 63

4.4.1 Training and Testing Classifiers 64
4.4.2 Feature Analysis 66
4.4.3 Generality of Trained Models 68
4.4.4 Evaluation on $D_{unknown}$ 69

4.4.5 Evaluation on D_{vt} . . . 69

4.5 Measurement and Discoveries . . . 72

4.5.1 Case Studies . . . 76

4.6 Discussion . . . 77
4.7 Conclusion . . . 78

5 ALL YOUR DNS RECORDS POINT TO US: UNDERSTANDING THE SECURITY THREATS OF DANGLING DNS RECORDS . . . 80

5.1 Introduction . . . 80
5.2 DNS Overview . . . 82
5.3 Dangling DNS Records . . . 84

5.3.1 Security Sensitive Dares . . . 85
5.3.2 IP in Cloud . . . 87
5.3.3 Abandoned Third-party Services . . . 90
5.3.4 Expired Domains . . . 91
5.3.5 Summary . . . 92

5.4 Measurement Methodology . . . 92

5.4.1 Domain Collection . . . 92
5.4.2 DNS Data Retrieval . . . 93
5.4.3 Searching for Dares . . . 93

5.4.3.1 Checking A Records (Lines 7 and 9) . . . 94
5.4.3.2 Checking Abandoned Services (Line 15) . . . 96
5.4.3.3 Checking Expired Domains (Lines 12 and 19) . . . 97

5.4.4 Limitations . . . 97

5.5 Measurement Results . . . 97

5.5.1 Characterization of Dares . . . 98
5.5.2 IP in Cloud . . . 100
5.5.3 Abandoned Third-party Services . . . 105
5.5.4 Expired Domains . . . 106
5.5.5 Exploiting Dares . . . 107

5.5.6 Ethical Considerations 108

5.6 Threat Analysis 108

5.6.1 Scamming, Phishing, and More 108
5.6.2 Active Cookie Stealing 110
5.6.3 Email Fraud 111
5.6.4 Forged SSL Certificate 111

5.7 Mitigations 112
5.8 Conclusion 114

6 PRACTICAL AND ROBUST DEFENSE AGAINST USE-AFTER-FREE EXPLOITS VIA CONCURRENT POINTER SWEEPING 116

6.1 Introduction 116
6.2 Background and Threat Model 118
6.3 Overview 119

6.3.1 High-Level Approach of pSweeper 119
6.3.2 An Illustration Example 120
6.3.3 Architecture of pSweeper 121

6.4 System Design 122

6.4.1 Memory Allocation Status Table 123
6.4.2 Locating Live Pointers 123

6.4.2.1 Pointers on Data Segment 123
6.4.2.2 Pointers on Stack 124
6.4.2.3 Pointers on Heap 124

6.4.3 Deferred Free 126
6.4.4 Concurrent Pointer Sweeping (CPS) 127

6.4.4.1 CPS Threads 127
6.4.4.2 Preventing Dangling Pointer Propagation 130

6.4.4.3 More pSweeper Threads 131

6.4.5 Object Origin Tracking (OOT) 132

6.5 Evaluation 133

6.5.1 Effectiveness of pSweeper 134
6.5.2 Performance on SPEC CPU2006 134

6.5.2.1 Runtime Overhead 136
6.5.2.2 Memory Overhead 138
6.5.2.3 Comparison to DangSan 139

6.5.3 Scalability on Multi-threaded Applications 139
6.5.4 Macro Benchmarks 140

6.5.4.1 Lighttpd 140
6.5.4.2 Mozilla Firefox 141

6.6 Discussion & Limitations 142
6.7 Conclusion 144

7 CONCLUSION **146**

BIBLIOGRAPHY **148**

LIST OF TABLES

3.1 Existing Methods to Detect and Confine Malicious PDF. 14

3.2 Runtime Behaviors Monitored in Two Contexts. 23

3.3 Confinement Rules 28

3.4 Methods provided in PDF to add scripts at runtime. 31

3.5 Dataset Used for Evaluation 32

3.6 Statistics of Static Features of Malicious Documents. 35

3.7 Parameter Configurations in Our System. 37

3.8 Detection Results 37

3.9 Comparison With Existing Methods 38

3.10 Execution Time (in seconds) of Static Analysis & Instrumentation. 40

3.11 Memory Overhead of Static Analysis & Instrumentation. 41

4.1 Training and test datasets. Columns 3∼4 include all domains we manually collected and thus some cells like those of $D_{unknown}$ do not have data. Columns 5∼8 present the number of domains obtained from two PDNS, Farsight and 360, respectively. 54

4.2 Sources of confirmed domain shadowing. 54

4.3 Features used in our approach to detect shadowed domains. Feature dimensions **D** and **C** denote Deviation and Correlation, respectively. Although some features use the same data source as previous work, e.g., resolution counts as in [141, 60], we model them in different ways. 58

4.4 List of top 50 popular subdomain names. 59

4.5 Importance of features. Features marked with an asterisk (*) are novel. 66

4.6 Security policy of the top 5 registrars in our detection. †Tucows is the owner of eNom and Hover etc. and provides services under them. . 74

5.1 Types of security-sensitive dangling DNS records. †Our work currently covers IPv4 only. ‡DNAME is semantically similar to CNAME, so we do not consider DNAME separately. 86

5.2 Summary of the attack vectors to which each type of Dare is vulnerable. 90

5.3 Evaluation set of domains. 93

5.4 Evaluated third-party services. 97

5.5 Statistics of distinct apex domains in `S` with confirmed Dares. † Some domains overlap across the above three lines (e.g., a domain has both Dare-A and Dare-CN). 100

5.6 Patterns of expired domains. 107

6.1 Real-world UaF vulnerabilities used for evaluation. 134

6.2 Detailed results on SPEC CPU2006 benchmarks. pSweeper runs at 1s sweeping rate. †M for million. 136

6.3 Overhead of pSweeper(-1s, -500ms, -nosleep) on three browser benchmarks. The percentage in parentheses is the slowdown. 142

6.4 Page load time (in seconds) of pSweeper on five popular websites. 143

LIST OF FIGURES

3.1 System Architecture 17

3.2 A Synthetic Sample of Malicious PDF. The start point can be object (2 0), (4 0), or (5 0). Any object can be selected as the start point, and here we assume (2 0) as the start point. 21

3.3 An Example to Illustrate Instrumentation 22

3.4 Workflow of Runtime Detection & Lightweight Confinement. 27

3.5 Two-stage Attack 31

3.6 Ratio of PDF Objects on Javascript Chain in Malicious and Benign Documents 33

3.7 Memory Consumption of Malicious and Benign Javascripts 33

3.8 Memory Consumption of PDF Reader When Opening Many Documents 34

4.1 Adding a subdomain in domain registrar GoDaddy. Assume the apex domain is foo.com. The added subdomain is `shadowed.foo.com` . . 48

4.2 Shadowed domains used in a campaign of EITest Rig EK in April, 2017. `app-garden.com` is a legitimate apex domain. 50

4.3 Workflow of `Woodpecker`. 52

4.4 Two sample records for subdomains under `Amazon.com` from 360 and Farsight (field explanation is covered in Section 4.3.1). 56

4.5 Performance comparison of classifiers under 10-fold cross-validation. The number of trees used in RandomForest is 100. All other classifiers use the default configuration in scikit-learn. 65

4.6 ROC of RandomForest on Farsight data when all, deviation-only (F1, F2, F4, F5, F7) and correlation-only (all others) features are used. . 67

4.7 ROC of RandomForest on Farsight data when features in a single category are used. 67

4.8 Performance of `Woodpecker` using RandomForest trained and tested on different PDNS sources. FS stands for Farsight. 68

4.9 Trend of domain shadowing. 73

4.10 Top 10 registrars in terms of distinct apex domains with shadowed domains. 73

4.11 Distribution of IPs in the top 10 countries. 74

4.12 CDF of the active days of shadowed domains. 75

5.1 The hierarchy of DNS. 83

5.2 The workflow of DNS resolution for `www.foo.com`. 84

5.3 Sample portion of a TLD zone file and DNS records on a resolver. For brevity, the TTL and class fields are omitted. 84

5.4 An example of a dangling A record. 85

5.5 Three paradigms of modern domain hosting. 88

5.6 aDNS setups for a domain hosted in the cloud. 88

5.7 Methodology overview. 92

5.8 Number of confirmed Dares for each dataset. 98

5.9 Number of potential Dares. 99

5.10 Categories of websites that have Dares. 101

5.11 Number of IP addresses milked on clouds over time. 102

5.12 Number of Dares on each third-party service. 103

5.13 Prices to register the expired domains for one year. 104

5.14 Authenticating Ephemeral IPs. 113

6.1 Illustration of pSweeper in time line. 121

6.2 The architecture of pSweeper. 122

6.3 Race conditions of pointers on stack. 124

6.4 Metadata of live objects. 127

6.5 Time of check to time of neutralization race. 129

6.6 Dangling pointer propagation in application threads. 130

6.7 Prevent dangling pointer propagation. Code snippets with a dark background are instrumented by pSweeper. 131

6.8 Use of pointer bits by OOT. 132

6.9 pSweeper's performance on SPEC CPU2006. 135

6.10 Dynamic instruction overhead and L1 data cache misses on SPEC CPU2006. 138

6.11 Memory overhead on SPEC CPU2006. 139

6.12 Overhead comparison with DangSan. 140

6.13 Scalability of pSweeper on PARSEC 3.0. The number of threads must be a power of two for `fluidanimate`. 141

6.14 pSweeper overhead on Lighttpd. 142

ABSTRACT

Numerous efforts have been devoted to securing computer systems in the past decade, which has rendered many previously popular attacks ineffective. In response to the arms race, cybercriminals are constantly seeking for new attack vectors. In this dissertation, we investigate several newly emerging attack vectors in cybercrime.

Our research first focuses on embedded malware inside Adobe PDF (Portable Document Format) documents. Due to its widespread use and Javascript support, PDF has become the primary vehicle for delivering embedded exploits since 2008. Unfortunately, existing defenses are limited in effectiveness, prone to evasion, or computationally expensive to be employed as on-line protection systems. To this end, we propose a context-aware approach for detection and confinement of malicious Javascript in PDF documents. Based on more than twenty thousand benign and malicious samples, our experimental evaluation shows that our defense system can achieve very high detection accuracy with minor overhead.

We further conduct the first comprehensive study on domain shadowing, a new strategy adopted by miscreants to build their attack infrastructures. We design a novel domain shadowing detector called `Woodpecker`, which characterizes shadowed domains based on a set of 17 novel features. By applying `Woodpecker` to the daily feeds of VirusTotal collected in two months, we can detect thousands of new domain shadowing campaigns. Our study highlights domain shadowing as an increasingly rampant threat since 2014.

Moreover, we discover a new security threat caused by dangling records in DNS. In a dangling DNS record (Dare), the resources pointed to by the DNS record are invalid, but the record itself has not yet been purged from DNS. Our work reveals that Dare can be easily manipulated by adversaries for domain hijacking. In particular,

we identify three attack vectors that an adversary can harness to exploit Dares. In a large-scale measurement study, we uncover 467 exploitable Dares in 277 Alexa top 10,000 domains and 52 edu zones, showing that Dare is a real, prevalent threat.

Finally, we present a novel defense called pSweeper to robustly protect against use-after-free (UaF) exploits with low overhead and pinpoint the root-causes of UaF vulnerabilities with one safe crash. The success of pSweeper lies in its two unique and innovative techniques: concurrent pointer sweeping (CPS) and object origin tracking (OOT). Unlike previous works that rely on pointer propagation tracking to find dangling pointers, CPS iteratively sweeps all live pointers in a concurrent thread to find dangling ones. OOT can help to pinpoint the root-causes by informing developers of how a dangling pointer is caused. We implement a prototype of pSweeper and validate its efficacy in real scenarios.

Chapter 1

INTRODUCTION

Computers have been indispensable tools in a modern society for more than half of century. Nowadays most computer systems are connected to the Internet, from mobile devices to high-end servers. Thus, protecting computer systems against malicious attacks is no longer an optional extra. For instance, it is reported that cybercrimes cost more than $400 billion to the global economy [159]. Over the last decade, numerous efforts have been devoted to securing computer systems against malware [181, 87, 89], phishing [219], spam [173], and software vulnerabilities [59, 151, 69, 90, 91, 177, 216]. These systems have greatly limited the effectiveness of traditional attack vectors like maliciously registered domains [114, 65, 74]. In response to this arms race, cybercriminals are constantly searching for new attack vectors. For example, one of the recently emerging threat, domain shadowing, can render existing domain reputation and blacklisting systems ineffective. To better protect end users, we need to design novel defenses against these newly emerging threats.

1.1 Problem Statements

In this dissertation, we aim to study several newly emerging attack vectors in cybercrimes. Specifically, we work on the following four problems.

(1) Malicious PDF document detection. Malware authors are constantly seeking new ways to compromise computer systems. Recently, they have embarked to take advantage of popular forms of data exchange, focusing their attention on malcode-bearing PDF documents [35]. This is clearly supported by the fact that the number of discovered PDF vulnerabilities has quadrupled in the last five years [55] with many attack cases having been reported [35, 187]. Despite the increasing number of successful

PDF infections and their impact on end users, thus far, only a few methods for detection of malicious PDF have been proposed as response to this emerging threat.

(2) Spawning malicious subdomains under legitimate domains. Domain names have been exploited by miscreants for illicit online activities for decades. In the past, miscreants mostly registered new domains for their attacks. However, the domains registered for malicious purposes can be easily deterred by existing reputation and proactive blacklisting systems. As a response to the arms race, miscreants have recently adopted a new strategy, called *domain shadowing*, to build their attack infrastructures. Specifically, instead of registering new domains, miscreants are beginning to compromise legitimate domains and spawn malicious subdomains under them. This has rendered almost all existing countermeasures ineffective and fragile, because subdomains inherit the trust of their apex domains (i.e., foo.com) and attackers can virtually spawn an infinite number of shadowed domains.

(3) Security threats caused by dangling DNS records. As one of the most critical components of the Internet, the Domain Name System (DNS) provides not only vital naming services but also fundamental trust anchors for accessing Internet services. Therefore, it has always been an attractive target to attackers [71, 125, 128]. In order to ensure the authenticity and integrity of DNS systems, tremendous efforts have been devoted to protecting both client and server mechanisms [84, 91, 177, 216]. In particular, a suite of security mechanisms like DNSSEC [69] have been deployed to secure the communication channels between DNS servers and clients. However, little attention has been paid to authenticating the integrity of the links between DNS servers and those resources to which DNS records point.

(4) Use-after-free vulnerabilities in software. Applications in C/C++ are notoriously prone to memory corruptions. With significant research efforts devoted to this area of study, the security threats posed by previously popular vulnerabilities, such as stack and heap overflows, are not as serious as before. Instead, we have seen the meteoric rise of attacks exploiting use-after-free (UaF) vulnerabilities in recent years, which root in pointers pointing to freed memory (i.e., dangling pointers). Although

various approaches have been proposed to harden software against UaF, none of them can achieve robustness and efficiency at the same time.

1.2 Contributions

The overall contributions of this dissertation lie in four different aspects to address the problems mentioned above, which are summarized as follows.

(1) Detecting and confining malicious Javascript in PDF. we introduce a context-aware approach to detect and confine malicious Javascript in PDF through static document instrumentation and runtime behavior monitoring. Our main contributions are:

- We propose static document instrumentation to achieve context-aware monitoring.
- We define five novel static features for characterizing the obfuscation techniques frequently used in malicious PDF.
- We conduct a series of experiments using a corpus of 18,623 benign and 7,370 malicious PDF documents. No false positive and few (25 out of 942) false negatives are generated during the evaluation.

(2) Automated detection of shadowed domains. We conduct the first comprehensive study of domain shadowing in the wild, and we present a novel defense system, called `Woodpecker`, to automatically detect shadowed domains. Our main contributions are:

- We identify and compose 17 features characterizing the usage, hosting, activity, and name patterns of subdomains.
- Five classifiers (Support Vector Machine, RandomForest, Logistic Regression, Naive Bayes, and Neural Network) are trained using these features. We achieve a 98.5% detection rate with an approximately 0.1% false positive rate under a 10-fold cross-validation when using RandomForest.
- We reveal that shadowed domains are involved in both exploit kits and phishing attacks.

- We observe that several domain shadowing cases exploit the wildcard DNS records, instead of algorithmically generating subdomain names.

(3) Unveiling and investigating a new threat in DNS. We unveil and investigate a largely overlooked threat in DNS: a dangling DNS record (Dare), which could be easily exploited for domain hijacking due to the lack of authenticity checking of the resolved resources. An adversary can exploit Dares to send arbitrary contents to users visiting the affected domains. Specifically, we make three contributions.

- We scrutinize the DNS specifications, during which four types of security-sensitive Dares are identified, including Dare-A, Dare-CN, Dare-MX, and Dare-NS. We present three attack vectors that an adversary can harness to hijack IP addresses and domain names in Dares.
- We then conduct a large-scale measurement to assess the magnitude of Dares in the wild. In total, 791 confirmed and 5,982 potential Dares are successfully found in our measurement study.
- We finally propose three defense mechanisms that DNS servers and third-party services can adopt to mitigate unsafe Dares.

(4) Practical and robust defense against UaF exploits. We design a novel defense system, called pSweeper, which effectively protects against UaF exploits, imposes low overhead for deployment in production environments, and pinpoints the root-causes of UaF vulnerabilities for easier and faster fixing. pSweeper follows the protection principle that potential UaF exploits can be disrupted by proactively neutralizing dangling pointers. Specifically, we make two contributions.

- We propose two unique and innovative techniques, concurrent pointer sweeping (CPS) and object origin tracking (OOT) to accomplish our design goals.
- We comprehensively evaluate the prototype of pSweeper with real-world vulnerabilities and various performance benchmarks.

1.3 Dissertation Organization

The rest of this dissertation is structured as follows. In Chapter 2, we discuss related work. In Chapter 3, we propose a context-aware malicious PDF detector. In Chapter 4, we propose `Woodpecker` to detect shadowed domains. Chapter 5 presents our study of dangling DNS records. In Chapter 6, we describe a novel defense against use-after-free vulnerabilities in software. Finally, we conclude the dissertation in Chapter 7.

Chapter 2

RELATED WORK

This chapter reviews related work in malicious document detection, DNS security, and use-after-free defense.

2.1 Malicious Document Detection

Existing research on malicious PDF detection has taken two directions, *static* methods which build statistical models from document content and classify unknown samples using machine learning, and *dynamic* methods which execute suspicious Javascript in some constrained environments.

Early static methods are based on n-gram analysis to detect universal malicious files [144, 190]. In 2011, Laskov et al. [137] presented PJScan, the first static method dedicated to the detection of malicious PDF. Using a patched SpiderMonkey, PJScan extracts lexical tokens of Javascript and trains an OCSVM (One Class Support Vector Machine) classifier to identify malicious PDF. Instead of analyzing Javascript, Malware Slayer [155] inspects the content of malicious PDF and counts the frequency of PDF keywords. Then, a set of keywords with high frequency are selected and fed into various machine learning algorithms for detection. PDFRate [194] extracts more structural features from PDF and thus builds a more accurate classifier. It can also detect targeted attacks. Srndic et al. [199] proposed a structural-path based method. They modeled a document as a set of structural paths and detected malicious PDF using Decision Tree and SVM (Support Vector Machine). Wepawet [41] uses JSAND [87], which leverages statistical and lexical features of Javascript, to detect malicious PDF. In general, static methods have been proven to be simple, fast, and effective. However, they are susceptible to mimicry attacks [154]. Our method differs from these fully static

methods in that, besides static features, we also use runtime behaviors of malicious Javascript for detection.

Compared with static detection, dynamic approaches are more robust against mimicry attacks. Tzermias et al. [210] proposed MDScan, which extracts Javascript from documents and executes it in instrumented SpiderMonkey and Nemu [180]. However, such a method suffers several limitations. First, it requires reliable Javascript extraction, which can be subverted by syntax obfuscations. Attackers can hide shellcode at some weird places in a document, e.g., in the title, and reference it in forms like "`this.info.title`". In this case, the extracted Javascript will fail to execute in emulated environments. Moreover, it is required to emulate PDF-specific Javascript objects, both documented [47] and undocumented like `printSeps()`. Finally, the proposed defense cannot be readily deployed on a user's system.

Meanwhile, malicious Javascript detection on the Web is a well-studied topic and many methods have been proposed [87, 89, 222, 227]. However, these methods are specially designed for detecting malicious Javascript on the Web and they are mainly based on the analysis of Javascript code itself. Differently, our approach monitors suspicious *system*-level behaviors in the context of Javascript execution.

Similar to our approach, CWSandbox [220] and PEB heuristics [181] also detect suspicious runtime behaviors of document readers. However, CWSandbox [220] is used primarily for detecting traditional malware, and it can be easily evaded by event-triggering or environment-sensitive malicious Javascript. Polychronakis et al. [181] proposed to execute shellcode in a CPU emulator and detect suspicious memory accesses using four heuristics. Egele et al. [105] presented a similar method which identifies potential shellcode at runtime and tests it in libemu [25]. Compared with these methods, we use different and more robust runtime features, which characterize the essential operations required in the infection process. Moreover, we neither identify shellcode, which can be evaded by using English Shellcode [158], nor emulate CPU, which is heavyweight. Snow et al. [195] proposed to monitor system call sequences of document readers. However, their method is context-free.

2.2 DNS Security

In past decades, significant research efforts have been devoted to studying the security of DNS. In the following, we provide an overview of previous works.

Cache Poisoning Attacks. Adversaries could exploit the flaws in a DNS server to inject incorrect entries, which will direct users to a different server controlled by adversaries. Since Bellovin [71] revealed this vulnerability in the 1990s, several mitigations [69, 90, 91, 177, 216] have been proposed. Adversaries can also tamper with DNS resolvers using spoofed DNS responses (i.e., off-path DNS poisoning [116, 115, 117, 128]). Instead of injecting entries to benign DNS servers or resolvers, more malicious resolution authorities have recently been deployed by adversaries [92, 135]. While having the same negative impact, our work is different from cache poisoning because we neither tamper with DNS resolutions nor set up malicious DNS services.

DNS Inconsistency and Misconfiguration. DNS is organized in a hierarchical tree structure, and it does not require strong consistency among nodes. Therefore, data changes in upper-level servers cannot override the cached copies in recursive resolvers. The outdated data will continue to serve users before reaching its TTL limit. The weak cache consistency could yield vulnerabilities like ghost domain names [125], i.e., the domains that have been deleted from TLD servers but still resolvable. Some approaches have been proposed to address this problem. For example, DNScup [84] proactively pushes changes in authoritative servers to recursive resolvers. However, in our study we reveal that inconsistency between DNS records and the connected resources is also prevalent and could pose serious security threats.

Pappas et al. [175] diagnosed three types of misconfigurations and found that these misconfigurations are widespread and degrade the reliability and performance of DNS. By contrast, our work uncovers a new type of DNS misconfiguration and studies its potential security threats. Although the issue of Dares has received some attention in two non-academic blogs [14, 20], our work is the first large-scale systematic study on this problem, in terms of DNS record types and the magnitude of Dares in high-value domains. We also identify two more vulnerable third-party services, one of which (i.e.,

Azure Cloud Service) cannot even be protected using domain ownership verification.

Security about domain ecosystem. The security issues in the domain ecosystem, including registrars and registries, have been studied for a long time. In particular, researchers investigated how domains are recruited and used by attackers for a spectrum of cybercrime businesses, like spam [63], exploit kit [109], blackhat SEO [102] and dedicated hosts [146]. Previous studies also show that adversaries actively register domain names similar to reputable ones (called typosquatting) in hopes of harvesting traffic from careless users [58, 202, 131]. When a domain is not serving its owner's website, the owner could leave it to a parking service that places ads there and share the revenue when the ads are viewed or clicked. However, the business practices of some parking services are problematic, as shown in previous studies [61, 215]. A recent study measures the security on the basis of individual TLD and demonstrates that the scale of free services on a TLD could impact its reputation [134]. Our study is complementary to these existing works in understanding the security issues in domain ecosystem.

2.3 Malicious Domain Detection

Detecting malicious domains. A wealth of research has been conducted on detecting malicious domains. Similar to our work, there are the approaches examining DNS data [65, 74, 66, 223, 67]. Shadowed domains exhibit different properties from the targets of previous studies. A new approach is needed and we show `Woodpecker` is capable of achieving the detection goal with a combination of deviation and correlation analysis. A recent work by Hao et al. [114] aimed to detect malicious domains at their registration time. Given that shadowed domains and their parent apex domains share the same registration information, such an approach is ineffective to detect shadowed domains. Plohmann et al. [179] and Lever et al. [142] conducted large-scale studies on malicious domains by running the collected samples in a sandbox environment. Botfinder [203] and Jackstraws [122] aim to detect C&C domains in botnets based on

the similar communication patterns of bot clients. By contrast, our approach does not assume the possession of any file samples (malware and web page).

Many works have focused on understanding and identifying malicious domain names. Jiang et al. [125] found that a malicious domain name could remain resolvable even long after it has been deleted from the upper level DNS servers or after its TTL has expired. Hao et al. [114] studied the domain registration behavior of spammers and found that spammers commonly re-register expired domains. Furthermore, Lever et al. [141] characterized the malicious re-registration of expired domains and demonstrated that the *residual trust abuse* is the root cause of many security issues. The Dares we studied in this dissertation can also be categorized as the residual trust abuse. The most salient difference is that Dares could be caused by not only the expired domains, but also a large number of subdomains, including those of the most well-known websites.

Detecting malicious web content and URLs. Detecting a malicious web page is another active research line in finding traces of cyber-criminal activities. Most of the prior works leverage web content and execution traces of a runtime visit for detection. Features regarding web content are deemed effective in detecting web spam [173], phishing sites [219], URL spam [204] and general malicious pages [81]. Malicious sites usually hide themselves behind web redirections but their redirection pattern is different from legitimate cases, which reveals themselves out [140, 218, 201]. Invernizzi et al. [121] showed that a query result returned from search engines can be used to guide the process of finding malicious sites. To trap more visitors, vulnerable sites are frequently compromised and turned into redirectors through code injection. Such a strategy introduces unusual changes to the legitimate sites, and can be detected by differing web content [77], HTTP traffic [62], and JS libraries [145]. The URLs associated with malicious web content might exhibit distinctive features, and previous works show machine-learning based approaches are effective to address this problem [152, 106, 153]. Obtaining web content or URLs usually requires active web crawling, which is time-consuming and ineffective when cloaking is performed by malicious servers. By contrast, our solution is lightweight and robust against cloaking.

2.4 Memory Safety in Software

Dangling pointer detection. Tools like Valgrind [168] and AddressSanitizer [188] track the (de)allocation status of each memory location. As long as a freed memory block is not reallocated, these tools can detect all dangling pointers. However, they can miss those pointing to a reallocated memory, which is common in UaF exploits. Another set of approaches extend each pointer with a unique identifier and check the validity on every pointer dereference [70, 166, 221, 224]. Unfortunately, software-only explicit pointer checks can slow applications by an order of magnitude. Recently, Nagarakatte et al. [163, 164] proposed a hardware-assisted approach that can provide full memory safety at low overheads. Undangle [80] detects dangling pointers by using dynamic taint analysis to track pointer propagations at runtime. It can serve as an in-house testing tool but not a runtime defense system.

Safe memory allocators. Cling [59] is a safe memory allocator that avoids memory reuse among objects of different types. It can thwart many, but not all, UaF exploits. DieHard [72] and DieHarder [171] are based on the idea of "infinite" heaps. Unfortunately, an infinite-heap is idealized but infeasible, and thus it can only provide probabilistic memory safety. Exterminator [172] extends DieHard to automatically fix dangling pointers by delaying object frees. Dhurjati et al. [97] used a new virtual page for each memory allocation and relied on page protection to detect dangling pointer accesses. By contrast, pSweeper proactively neutralizes all dangling pointers.

Garbage collection (GC). GC [16, 76] not only heads off exploits but prevents program crashes due to UaF vulnerabilities. However, most GC algorithms can consume more memory because they defer free until there is insufficient memory or applications explicitly ask. By contrast, pSweeper can free memory after a round of pointer sweeping. Moreover, stop-the-world GC can cause unpredictable interference to application performance. Finally, although pSweeper can only probabilistically mask program crashes, it guarantees to pinpoint the root-causes of UaF vulnerabilities when programs crash.

Safe C languages. Fail-safe C [174] implements a completely memory-safe

compiler that is fully compatible with ANSI C. It uses garbage collection to protect against dangling pointers. There are also safe C dialects, such as Cyclone [110, 126] and CCured [86, 167]. Although they attempt to keep compatible with C/C++ specifications, non-trivial efforts are still needed to retrofit legacy programs.

Parallelizing security checks. Concurrent security checks as in pSweeper have also been adopted in several previous works. Speck [169] decouples security checks from applications and executes them in parallel on multiple cores. Unlike Speck, pSweeper does not use speculative execution. Cruiser [228] and Kruiser [206] use concurrent threads to detect buffer overflows in user applications and kernels, respectively. ShadowReplica [124] accelerates dynamic data flow tracking by running analysis on spare cores. However, pSweeper tackles a different problem and faces unique challenges.

Chapter 3

DETECTING MALICIOUS JAVASCRIPT IN PDF THROUGH DOCUMENT INSTRUMENTATION

3.1 Introduction

Malware authors are constantly seeking for new ways to compromise computer systems. Recently, they have embarked to take advantage of popular forms of data exchange, focusing their attention on malcode-bearing PDF documents [35]. The PDF standard has several unique advantages when used as an attack vector: (1) it has replaced Microsoft Word as the most dominant document format; (2) it has been widely considered to be safe; (3) it is easy to craft a malicious PDF; and more importantly, (4) it supports Javascript. All of these features have made PDF one of the most attractive exploitation vehicles. This is clearly supported by the fact that the number of discovered PDF vulnerabilities has quadrupled in the last five years [55] with many attack cases having been reported [35] [187]. The most striking observation comes from Microsoft malware protection center, showing that the exploitation of old PDF vulnerabilities is on the rise [35].

Despite the increasing number of successful PDF infections and their impact on end users, thus far, only a few methods for detection of malicious PDF have been proposed as response to this emerging threat. Unfortunately, it appears that traditional signature and behavior based detection methods, which are favored by the majority of modern anti-virus software, cannot handle malicious PDF well. Recently, researchers exploit the structural differences between benign and malicious documents to detect malicious PDF [194] [199] [155] [137]. These methods have been proven to be simple, fast, and accurate. However, when attackers are aware of these static features, they can evade easily [154]. Another recent work extracts and tests malicious Javascript in

Table 3.1: Existing Methods to Detect and Confine Malicious PDF.

Method	Difficult to Evade	End-Host Deployment	Need Emulation	Low Overhead
Signature	No	Yes	No	Yes
Structural [199] [194] [155]	No	Yes	No	Yes
Extract-and-Emulate [210]	Neutral	No	Yes	No
Lexical Analysis of Javascript [137]	Neutral	Yes	No	Yes
Adobe Sandboxing [51]	Neutral	Yes	No	Yes
CWSandbox [220]	Neutral	No	Neutral	No
Our Method	Yes	Yes	No	Yes

an emulated interpreter [210]. Although it is more robust against evasion, attackers can still exploit syntax obfuscations to subvert Javascript extraction. Also it is very costly to emulate all PDF-specific Javascript objects. In 2009, Adobe announced the Protected Mode, a sandboxing mechanism that runs PDF reader in a confined environment. Although it raises the bar, Adobe Sandbox has its own drawbacks. An obvious one is that there exist vulnerabilities in the sandbox itself. Actually hackers have already discovered different ways to escape Adobe Sandbox [150] [217].

The detection of malicious PDF exhibits two distinct challenges. First, users tend to open multiple PDFs simutaneously. However, the runtime behaviors of a PDF reader can vary as different documents are opened, and both benign and malicious PDFs are processed by one single thread in the PDF reader. These can inevitably affect detection accuracy due to the interference among multiple open documents. Second, although it is straightforward to locate traditional malware once detected, it is non-trivial to pinpoint these malicious PDF documents since all open documents could be malicious.

In this dissertation, we introduce a *context-aware* approach to detect and confine malicious Javascript in PDF through static document instrumentation and runtime behavior monitoring. Our method is motivated by the fact that some essential operations of Javascript in malicious PDF rarely occur in benign documents. Our context-aware approach can efficaciously overcome the aforementioned two challenges. On one hand, context-aware approach can make detection features, like suspicious memory consumption, more effective in detection. On the other hand, the context information explicitly

indicates which open documents are malicious.

There are different ways to achieve context-aware monitoring. One intuitive choice is to extract Javascript from documents [210, 87]. Alternatively, Javascript interpreters can be instrumented [89]. But these methods are neither robust nor easy to implement in practice. Instead, we choose to perform *static document instrumentation*. This method, to the best of our knowledge, has never been explored before for PDF malware detection and confinement. For each PDF Javascript snippet, we include a prologue and epilogue to inform our runtime detector for the entry to and exit from Javascript context. The advantage of using static document instrumentation over the other two alternatives lies in three aspects. First and most important, it is immune to code and syntax obfuscations. Second, it does not need to emulate Javascript interpreters, resulting in much less development effort and minor computational overhead. Last but not least, it provides good portability and can be easily deployed at end hosts.

When an instrumented document is loaded, our runtime detector monitors the behaviors of a PDF reader process and identifies potential infection attempts from Javascript. The infection attempt manifests itself through a sequence of suspicious actions, such as exploiting to compromise systems, retrieving malware and executing it. By monitoring these suspicious behaviors as evidence of infection, we compute a weighted sum to detect malicious PDF.

Our system also defines five novel static features for detection. These features characterize the obfuscation techniques frequently used in malicious PDF. The combination of static and runtime features will be more effective and robust than existing methods, which are either fully static [199, 194, 155] or fully dynamic [210, 220]. A more thorough comparison between our method and others is presented in Table 3.1.

For any new intrusion detection mechanism, we need to perform a security analysis—a task that in many cases is even more important than its detection performance. In principle, it is required that the defense system remains robust and secure even when its internal operation is exposed to attackers. To this end, we conduct a security analysis of our approach showing that our system is still effective in detection

and robust against evasion attacks even in the presence of a sophisticated adversarial environment. In particular, a list of potential advanced attacks are discussed and mitigations for their impact are presented.

To validate the efficacy of our system, we conduct a series of experiments using a corpus of 18623 benign and 7370 malicious PDF documents. The experimental results show that our static and runtime features achieve very promising detection performance. No false positive and few (25 out of 942) false negatives are generated during the evaluation. It takes only 0.04 seconds on average to instrument a malicious sample and about 5.5 seconds to process a very large (20 MB) document. The slowdown caused by our runtime detector is 0.093 seconds for a single Javascript. Even when as many as 20 separate scripts are instrumented, the slowdown does not exceed 2 seconds. Overall, our system provides an effective defense against malicious PDF in practice.

3.2 System Design

3.2.1 Architecture

Our system consists of two major components, front-end and back-end, working in two phases. In Phase-I, the front-end component statically parses the document, analyzes the structure, and finally instruments the PDF objects containing Javascript. Then, in Phase-II when an instrumented document is opened, the back-end component detects suspicious behaviors of a PDF reader process in context of Javascript execution and confines malicious attempts. Figure 3.1 shows the architecture of our system.

Static Analysis and Instrumentation: For suspicious PDF, the front-end first parses the document structure and then decompresses the objects and streams. A set of static features are extracted in this process. When a document has been decompressed, the front-end will instrument it and add *context monitoring code* for Javascript. In some cases, if the document is encrypted using an owner's password, i.e., a mode of PDF in which the document is readable but non-modifiable, we need to remove the owner's password. With the help of PDF password recovery tools like [15], this can be done easily and very fast.

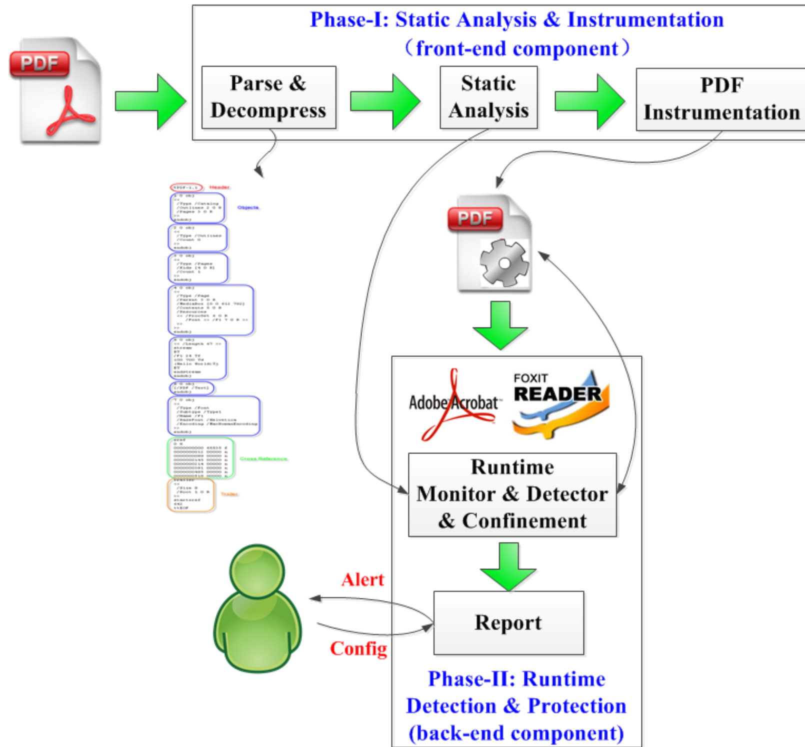


Figure 3.1: System Architecture

Runtime Detection: The back-end component works in two steps, runtime monitoring and runtime detection. When an instrumented PDF is loaded, the context monitoring code inside will cooperate with our runtime monitor, which tries to collect evidence of potential infection attempts. When Javascript executes to the end or a critical operation occurs, the runtime detector will compute a malscore. If the malscore exceeds a predefined threshold, the document will be classified as malicious.

3.2.2 Static Features

Several recent works have proposed to detect malicious PDF by statically analyzing document content [194] [199] [155]. Static methods are simple, and they have promising performance in detecting existing malicious documents. In this work, we define five novel static features to aid runtime detection by leveraging the obfuscation techniques used in malicious PDF. Although static features are vulnerable to evasion,

their usefulness for detection lies in two aspects: (1) if malicious documents use obfuscations, our system can detect them with higher confidence; and (2) if not, then the unobfuscated documents can be processed more easily and accurately by our front-end component. In the following, we detail the static features used in our system.

Ratio of PDF Objects on Javascript Chain: In PDF, a labelled object is called an indirect object, which can be referred to by other objects [46]. Sometimes, there are several indirect objects between the root and the one containing real data. These PDF objects form a reference chain. In the sample PDF as shown in Figure 3.2, there are ten indirect objects. We extract every chain containing at least one Javascript object on the path. We call it a *Javascript chain*. This feature computes the ratio of the objects involved in Javascript chains to the total objects in a document. Normally, malicious documents contain few data and many of them have only one blank page. Thus, in malicious documents, the ratio should be relatively high.

PDF Header Obfuscation: The PDF specifications require only that the header appears somewhere within the first 1,024 bytes of the file [46]. Benign documents rarely have incentives to obfuscate PDF header, but malicious documents are more willing to do so. Actually a recent work has proposed to manipulate the file type identifiers to evade anti-virus software [123]. Another trick attackers can use is to specify an invalid version number in header. Our system checks if PDF header appears at the very beginning of a document and if the header format is valid.

The following three features are checked for objects on Javascript chains only.

Hexadecimal Code in Keyword: PDF standard allows any character except NULL to be represented by its 2-digit hexadecimal code, preceded by one or more number signs (#). Many malicious documents use this trick to hide keywords. For example, in object (4 0) in Figure 3.2, `/JavaScript` is encoded as `/JavaScr##69pt`.

Count of Empty Objects: Object (6 0) in Figure 3.2 shows a Javascript chain from a malicious PDF. In this document, the Javascript chain ends with an

empty object. Actually, the real malicious Javascript is embedded in another chain. Our system counts the number of empty objects in a document.

Levels of Encoding: Encoding in PDF is used primarily for compression. Normally benign documents use only one level of encoding since multi-encoding brings little improvement. However, malicious documents tend to use multiple levels to evade anti-virus software.

Our system records the maximal encoding levels used on Javascript chains. Maximum, rather than average, is used for two reasons: on the one hand, maximum is more effective; on the other hand, average is susceptible to mimicry attacks. For example, attackers can deliberately insert many Javascript chains with one level of encoding. In this case, the average drops close to one.

3.2.3 Document Instrumentation

Due to its wide-spread adoption, simplicity, and strong expressiveness, Javascript is employed by the vast majority of malicious PDFs in the wild. Therefore, identifying and confining malicious Javascript in PDFs can effectively mitigate the risk they currently pose to Internet users. Motivated by the fact that malicious Javascript behaves significantly different from the benign one in system-level, we propose a context-aware detection and confinement approach. The core idea is to confine operations that are deemed suspicious based on the context of Javascript execution.

In order to implement the context-aware approach, one of the challenges is to identify when Javascript starts to execute and when it finishes. A simlpe solution is to extract Javascript from documents and execute it in an emulated environment. However, the extract-and-emulate method cannot guarantee reliable Javascript extraction, as demonstrated by an example shellcode in object (4 0) in Figure 3.2. Moreover, it can be very computationally expensive to emulate PDF-specific objects. An alternative option is to instrument a Javascript interpreter. For example, a snippet of monitoring code can be inserted at the entry and exit points of the Javascript interpreter. Although easy to implement, we do not choose this approach for two reasons. First,

interpreter instrumentation is insecure and can be easily bypassed. Second, interpreter instrumentation has poor portability.

To overcome the aforementioned limitations, we propose to leverage static document instrumentation, which requires neither Javascript extraction nor environment emulation. Using our approach, a snippet of context monitoring code is inserted into the document statically. Every time Javascript gets executed and finishes execution, the context monitoring code takes control and informs our runtime detector.

The first step of our method is to reconstruct all Javascript chains in a document. We use a similar technique described in previous works [137] [154] [210] to locate Javascript. Specifically, we scan the document for keywords `/JS` and `/JavaScript` that indicate a string or stream containing Javascript [46]. Next, we recursively backtrack to find the ancestors on a chain and forward search for the descendants. At the end of this process, we can extract a collection of Javascript chains. We only instrument the chains associated with some triggering actions, such as `/OpenAction` and `/AA`. Figure 3.2 illustrates the execution steps of the aforementioned algorithm. This algorithm is quite robust since it is immune to Javascript code obfuscation, and according to [46], the keyword `/JavaScript` should be plain text.

Javascript in PDF can be invoked either singly or sequentially (through `/Next` and `/Names`). The instrumentation process for single Javascript is shown in Figure 3.3. We first store the original code in a string which is passed as argument to `eval()` and then we prepend and append our context monitoring code to it. This process is quite simple and does not require sophisticated code analysis. The only operation we perform is to scan the code and add '\\' for ′ and ″ in the original Javascript code. When Javascript snippets are triggered, the context monitoring code, rather than the original script, gets executed first and it informs the runtime detector of the entrance and exit of Javascript context. During this process, the context monitoring code has to be able to communicate with the runtime detector. PDF provides three possible channels for communication: shared file, HTTP, and SOAP (Simple Object Access Protocol). Shared file is inefficient and insecure. The `Net.HTTP` method can

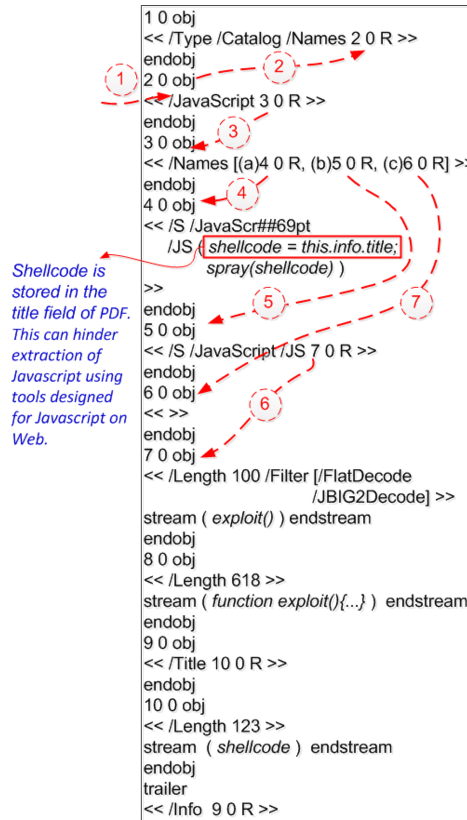


Figure 3.2: A Synthetic Sample of Malicious PDF. The start point can be object (2 0), (4 0), or (5 0). Any object can be selected as the start point, and here we assume (2 0) as the start point.

be invoked only outside of a document [47], i.e., cannot work in our context monitoring code. We select SOAP for our implementation to avoid the pitfalls of the other communication options. To achieve that, a tiny SOAP server is built into the detector enabling the communication with the context monitoring code synchronously. A randomly generated key is used to protect the SOAP communications. The key has two parts, Detector ID and Instrumentation Key. Detector ID is generated when our system is installed. In case that an already instrumented document is downloaded, this field can be used to filter out communications from the invalid context monitoring code. The second field is randomly generated when instrumenting a document and it uniquely identifies an instrumented document. We also maintain a mapping between instrumented document and key. When instrumenting a file, we first ensure that no

duplicate instrumentation is carried out on a single document. We further discuss the security of the key in Section 3.3.

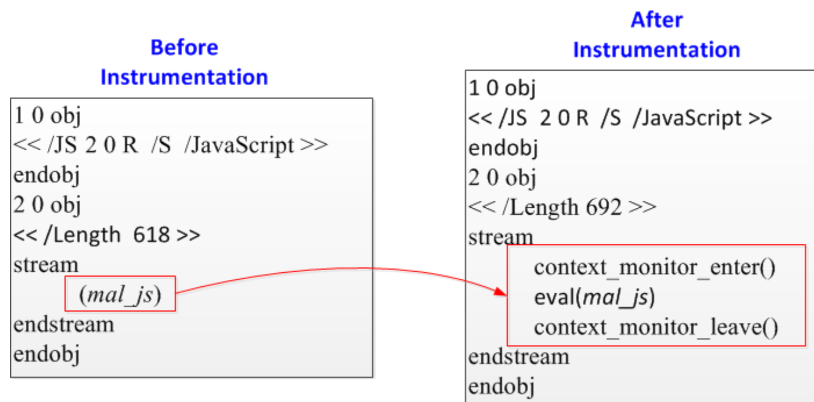


Figure 3.3: An Example to Illustrate Instrumentation

For sequentially invoked scripts, the process is a little different. We can simply insert the context monitoring code for each separate Javascript listed in `/Names` dictionary or `/Next` field. However, this can incur intolerably high overhead. A better choice is to parse the chain and enclose all scripts invoked sequentially using one single context monitoring code, which is taken in our system.

Finally, attackers can also dynamically add Javascript using the methods listed in Table 3.4 and delay the execution of Javascript using `setTimeOut()`. The two cases are specially handled in Section 3.3.

3.2.4 Runtime Features

When an instrumented PDF is opened, our stand-alone detector starts to monitor suspicious behaviors of the PDF reader and collect evidence of infection. We detect those essential operations that compromise target systems.

To improve the chance of successful exploits given various modern security enhancements, heap spraying has become the preferred weapon in hackers' arsenal. When heap is sprayed, a vulnerability like CVE-2008-2992 can be triggered to transfer the control to shellcode, which will execute the dropped malware, carry out drive-by-download,

or establish a reverse bind shell. All of these operations should rarely occur in benign Javascript. Thus, any occurrence of these operations in the context of Javascript execution can be considered as suspicious. This is referred to as *JS-context* monitoring. In addition, we note that unlike browsers which normally work in multi-thread, PDF readers process documents in single-thread. That is, during the execution of Javascript, no other PDF objects in the same or another document will be processed. This fact simplifies our method and we do not need to consider the potential false positives caused by concurrency.

JS-context monitoring can effectively detect malicious documents that exploit the vulnerabilities in Javascript interpreters. However, attackers can also exploit other vulnerabilities like CVE-2010-3654 in Flash and CVE-2010-2883 in CoolType.dll. Javascript in such malicious documents is normally responsible for heap spraying and malformed data crafting. In such cases, probably the JS-context monitoring can detect only one suspicious operation, i.e., heap spraying, which is insufficient for accurate detection. To complement JS-context monitoring, we also monitor the runtime behaviors after Javascript finishes (*out-JS-context*).

Table 3.2 lists the runtime behaviors we monitor in the two contexts above. Each monitored behavior is defined as one runtime feature in our system. Essentially, these behaviors are modeled as sequences of system calls. While using system calls to detect anomaly is not new [195] [108] [133], our method differs in two aspects. First, most previous works focus on detecting the behavior deviations from expected execution. But we detect the infection attempts of malicious code. Second, although there exist works on modeling the behaviors of malware [133], our method relies on the context-aware monitoring which has not been explored in previous works. Below, we continue to explain the details of each monitored behavior.

Table 3.2: Runtime Behaviors Monitored in Two Contexts.

Context	Runtime Behaviors
Out-JS-Context	Process Creation and DLL Injection
JS-Context	Memory Consumption, Network Access, Mapped Memory Search, Malware Dropping, Process Creation, and DLL Injection

Malware Dropping: A common practice of malicious PDF is to drop some malware to a user's file system. To monitor the malware dropping, we hook the APIs `NtCreateFile()`, `URLDownloadToFile*()`, and `URLDownloadToCacheFile*()` on Windows.

Suspicious Memory Consumption: In heap spraying, malicious code fills the heap with a NOP sled appended with shellcode. Subsequently, it attempts to divert the control flow to any address covered by the NOP sled that leads to the shellcode execution. In an effort to increase the probability of hitting a NOP, malicious code attempts to write a large area of memory, usually more than 100 MB [196].

Suspicious memory consumption can be very promising in detecting the presence of heap spraying, especially if monitored in JS-context. The context-free monitoring can cause many false positives, e.g., in a case that many documents are opened simultaneously. However, the context-aware monitoring in our method can effectively eliminate most noise. We check the `PROCESS_MEMORY_COUNTERS_EX` structure [56] at the entry/exit of JS context and when other in-JS sensitive APIs are captured.

Suspicious Network Access: Unlike on the Web, Javascript in PDF rarely connects to the Internet and its primary function is to dynamically render a document, which rarely relies on network communications. Actually, the number of Javascript methods provided in PDF for network access is limited and most of them can be used only in restricted conditions. For example, `app.mailmsg()` and `app.launchURL()` establish network connections using third-party applications (email clients and browsers), which are not monitored by our runtime detector. And, the `Net.HTTP` object cannot be invoked by Javascript embedded in a document. Thus, any network connection generated in JS-context should be considered as suspicious. In our system, we hook all `connect` and `listen`. Note that we white-list the communications between the runtime detector and the context monitoring code.

Mapped Memory Search: Besides drive-by-download, attackers can also embed malware in a document. Such a technique is called Egg-hunt. In [185], a malicious

sample using egg-hunt is analyzed. One challenge of egg-hunt is that attackers cannot know where malware is loaded in memory and they have to search the whole address space. However, some memory in the address space is unallocated, and dereferencing it can lead to segmentation fault. In order to prevent access violations, attackers have to employ some techniques to safely search the virtual address space. Several effective techniques, for both Linux and Windows, are described in [192]. In our implementation, `NtAccessCheckAndAuditAlarm()`, `IsBadReadPtr()`, `NtDisplayString()`, and `NtAddAtom()` are monitored.

Process Creation: The final step of an attack lies in execution of the dropped malware. Attackers can create a new process to execute the malware. In JS-context, this behavior can be a strong sign of infection attempt; while in out-JS-context, it can cause false positives. We observe that Windows error report programs and tools distributed with PDF readers, which obviously are benign, are usually invoked. So, we add them to a white-list. In implementation, we monitor `NtCreateProcess()`, `NtCreateProcessEx()`, and `NtCreateUserProcess()`.

DLL Injection: In the wild, usually attackers prefer to execute malware via DLL injection. This behavior should never occur in JS-context and rarely occur outside of JS-context. Thus, we monitor DLL injection in both JS-context and out-JS-context. In implementation, we monitor `CreateRemoteThread()`.

3.2.5 Runtime Detection and Confinement

Detection. The workflow of runtime detection and lightweight confinement is shown in Figure 3.4. The runtime detector works in three steps. Initially, all sensitive operations are ignored until at least one in-JS operation is captured from an unknown PDF. Although it may cause false negatives to discard out-JS operations at this step, we believe it is worthwhile for achieving a lower false positive rate and higher performance. Next, the detector starts to continuously record all sensitive operations. The core logic

of the runtime detector is a weighted sum, as shown in Equation 3.1.

$$malscore = w_1 \sum_{i=1}^{7} F_i + w_2 \sum_{i=8}^{13} F_i. \tag{3.1}$$

The first part represents the static and out-JS features. The second part denotes the in-JS features. The features are numbered from 1 to 13, and the runtime features are numbered in the order they appear in Table 3.2. All these features are normalized to binary values. Instead of assigning a weight for each feature, we set a weight for each "part" in the equation. We also define a threshold and if the malscore exceeds it, the document is tagged as malicious. The feature normalization, weight and threshold setting are based on the statistical results of a large corpus of benign and malicious samples. We provide a detailed description in Section 3.4.3.

In real world, users usually open many PDFs simultaneously, which must be correctly handled by the runtime detector. For each unknown open PDF which has carried out at least one in-JS operation, we maintain a separate malscore and a set of related operations. In-JS operations affect the corresponding malscore only, while out-JS operations contribute to every active malscore. Finally, in order to handle the case that multiple malicious PDFs work together to attack stealthily, we maintain a list of executables downloaded in JS context. When an in-JS operation invokes an executable in the list, we intentionally prepend a malware dropping operation for this PDF and append a malware execution operation for another PDF that downloads the file. Malscore is volatile, implying that it no longer exists when a PDF reader is closed. However, the maintained list of executables is persistently stored. When an alert is raised, we report the malscore, associated features, and the detected malicious PDFs to users.

Confinement. In Figure 3.4, the operations enclosed in solid border are confined. Our lightweight confinement, as well as runtime monitoring, is based on Windows API hooking. There are various ways to implement API hooking, e.g., modifying the

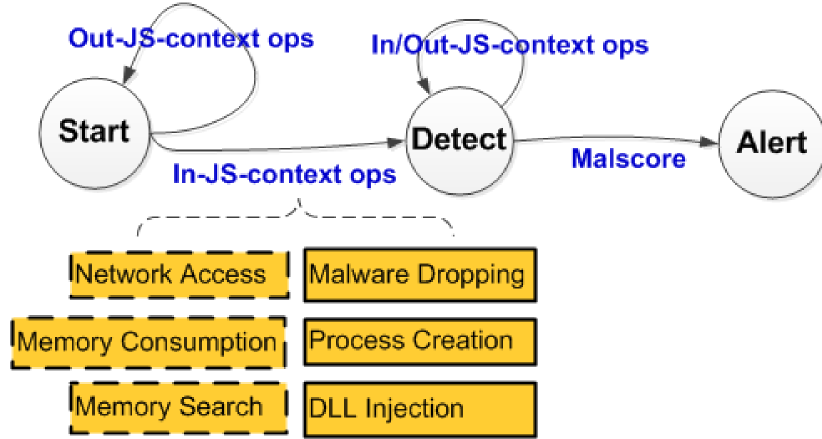


Figure 3.4: Workflow of Runtime Detection & Lightweight Confinement.

system service dispatch table (SSDT) or the interrupt descriptor table (IDT). Our prototype adopts the import address table (IAT) hooking since it is simple, effective, and efficient. Although attackers could leverage `GetProcAddress()` or call kernel routines directly to bypass IAT hooking, it is quite uncommon [220]. In the future, we will use advanced kernel mode hooks to make it more difficult to evade.

An essential step of IAT hooking is to inject our hook DLL. There are two popular implanting techniques on Windows, i.e., remote thread injection and AppInit registry modification [43]. Our prototype adopts the latter approach. As AppInit modification can affect the whole system, which is undesirable, we utilize a similar technique introduced in [28]. The basic idea is to develop a trampoline DLL, which further loads the IAT hook DLL if the host process is a PDF reader and otherwise does nothing. In this way, our confinement affects PDF readers only and thus incurs negligible overhead to the whole system.

Moreover, since API hooks execute in a PDF reader process, we need a channel for communications between API hooks and our stand-alone runtime detector. In our prototype, TCP socket is used. When the hook DLL is injected, its first job is to set up a TCP connection to the runtime detector. At runtime, it sends the captured API, API parameters, and memory usage (for suspicious memory consumption in §3.2.4) to the runtime detector.

Table 3.3 shows the pre-defined confinement rules executed by the runtime detector and Hook DLL. The rules are quite straightforward. The only issue that deserves attention is, in order to confine the created process, we use an existing sandbox tool, Sandboxie [36]. Currently, we just handle three sensitive operations. However, we can easily extend existing confinement rules.

Table 3.3: Confinement Rules

Operation	Rules	
	Execute In Hook DLL	**Execute In Runtime Detector**
Malware Dropping	Before alert, call original API.	Before alert, maintain the list of downloaded executables; When alert, isolate.
Process Creation	Before alert, reject the call since it will be invoked by runtime detector.	Before alert, run target program in Sandboxie [36]; When alert, terminate and isolate the program.
DLL Injection	Always reject.	Isolate the injected DLL.

3.2.6 De-instrumentation

In reality, it is common to open a document many times. In order to improve performance and scalability, we can monitor new documents only. We adopt an intuitive and simple approach, *document de-instrumentation*, to achieve this goal. When a document is identified as benign, our system removes the context monitoring code from it, i.e., de-instrumenting it. De-instrumentation is done in background after the PDF reader is closed. To facilitate de-instrumentation, our static instrumentation component will generate and export the corresponding de-instrumentation specifications when instrumenting a document. De-instrumentation significantly improves scalability while no security hole is introduced. Note that de-instrumenting at-once is a simple heuristic. A configurable parameter and randomization can be introduced to set the number of opens before de-instrumentation.

3.3 Security Analysis

For any intrusion detection system, it is a must to enforce its own integrity and security. In this section, we first describe the threat model. Then, we present a list of potential advanced attacks and our countermeasures.

3.3.1 Threat Model

In our analysis, we assume an advanced attacker who can access our code and test it for unlimited times. Moreover, the attacker can embed some arbitrarily large shellcode in the document. The shellcode is able to: (1) identify the heap, stack, and code areas in memory; (2) scan the whole virtual address space; and (3) modify any memory content.

Meanwhile, we also assume that attackers can neither (1) understand the meaning of data in memory if there is no identifiable signature nor (2) manipulate our static instrumentation code since the instrumentation component gets executed before malicious code.

3.3.2 Potential Advanced Attacks and Countermeasures

Mimicry Attack: An obvious attack is the mimicry attack, targeting the messaging mechanism between the context monitoring code and the runtime detector. Attackers try to steal the key used in communications and send a fake message to the runtime monitor, mimicking the epilogue of the context monitoring code. Then, the shellcode can do anything without monitoring. An alternative approach is to search for our episode code and execute it before carrying out malicious operations. We argue that

> *our random key, context monitoring code randomization and duplication, and zero tolerance to fake message can effectively defeat such a mimicry attack.*

Attackers can use either signature-based [94] or test-based [111] methods to search for keys in memory. In many cases, the key is stored at some fixed addresses or somewhere near an identifiable string, e.g., "auth-password" or "MyPwd". Such a signature remains intact once software is released, and hence attackers can easily locate the key in memory. Our system avoids generating signatures through: (1) executing the context monitoring code using `eval()`; (2) generating the key randomly during

static instrumentation; (3) randomizing the structure of the context monitoring code; and (4) creating copies of fake context monitoring code.

It is much easier to defeat the test-based cracking. We enforce that whenever a fake message is received, we tag the active document as malicious. Note that attackers cannot launch DoS attacks by pretending to be another PDF. As mentioned before, PDF readers work in single-thread and only one document is active at any time. From the key in the prologue, we can identify the active document, which is responsible for the fake message.

Runtime Patching Attack: Attackers can also carry out the runtime patching attack. There are two separate scripts in the document, so we instrument each of them independently. When the shellcode in the first script gets executed, it can locate the second script in memory and patch out the context monitoring code. Then, the second script can execute without monitoring. A variant attack is to distribute malicious Javascript in two separate documents.

To avoid the runtime patching attack, we ensure to take control at the beginning of each script. We apply encryption to enforce such control retaining. During instrumentation, an encryption scheme is randomly selected to encrypt the original script, and the decryption method is embedded in the prologue of the context monitoring code. In this way, malicious Javascript cannot get executed without our context monitoring code.

Moreover, several obfuscation methods are used to make it impossible for attackers to eliminate the context monitoring code but still keep the decryption code.

Staged Attack: An advanced attacker can split the exploit into multiple stages. Let us consider the simplest two-stage attack, as shown in Figure 3.5. In step 3, the Stage_2 code can be installed using Javascript methods listed in Table 3.4.

To defeat this kind of attack, we analyze the Javascript code and search for the methods in Table 3.4 during static instrumentation. Then, we instrument the dynamically added scripts that are stored in the parameters of these methods. A more

1) Instrument the target PDF.
2) Context monitoring code informs the enter of Javascript.
3) The Stage_1 shellcode setups Stage_2 code at runtime.
4) Context monitoring code informs the leave of Javascript.
5) Stage_2 shellcode is triggered by some event later.

Figure 3.5: Two-stage Attack

Table 3.4: Methods provided in PDF to add scripts at runtime.

Method	**Trigger Event**
Doc.addScript()	Open the document
Doc.setAction()	Close/Save/Print the document
Doc.setPageAction()	Open/Close a page
Field.setAction()	Operate on a form field
Bookmark.setAction()	Click the bookmark

robust solution we are working on is to hook these methods in Javascript interpreters and instrument dynamically inserted scripts on-the-fly. Since we only need to hook five methods, the development efforts and runtime overheads should be minor.

Delayed Execution: Another evasion approach is to delay the execution of Javascript. This can be achieved through `app.setTimeOut()` and `app.setInterval()` [47]. Our countermeasure is similar to the one for staged attack and we intentionally instrument the two Javascript methods above.

3.4 Evaluation

To validate the efficacy of our proposed approach, we implement a prototype on Windows. The front-end component is implemented in Python 2.7. The runtime monitor and detector in the back-end component are implemented in C and Java, respectively. And, the tiny SOAP server in the runtime monitor is built using the Web service framework JAX-WS. Based on a large corpus of real data, we first evaluate

the effectiveness of our detection model and then examine the runtime overhead of our prototype.

3.4.1 Data Collection

We collected more than twenty thousand benign and malicious samples for this study. Table 3.5 summarizes the dataset used in our evaluation. The benign documents are from four trusted sources: (1) we collected thousands of documents from two users' file systems; (2) we downloaded hundreds of official forms and reports from large organizations like governments and well-known companies; (3) we collected a set of non-malicious PDF files from Contagiodump [54]; and (4) we randomly crawled over ten thousand of documents using Google and tested them using anti-virus software. The malicious samples are from Contagiodump and those containing no Javascript are excluded.

Table 3.5: Dataset Used for Evaluation

Category	# of Samples	# with Javascript	Size
Known Benign	18623	994	11.84 GB
Known Malicious	7370	7370	172 MB
Total	25993	8364	12.01 GB

3.4.2 Feature Validation

Before measuring detection accuracy, we first validate the capability of our detection features to distinguish between benign and malicious documents. Here we present the statistical results of the features used in our system.

Static Features: We scanned all benign documents and found 994 samples containing Javascript. The following evaluation mainly relies on these 994 samples.

The first static feature we validate is the ratio of PDF objects on Javascript chains. Figure 3.6 shows the cumulative distribution function of the ratio in benign and malicious documents. As we can see, about 95% of malicious documents have a

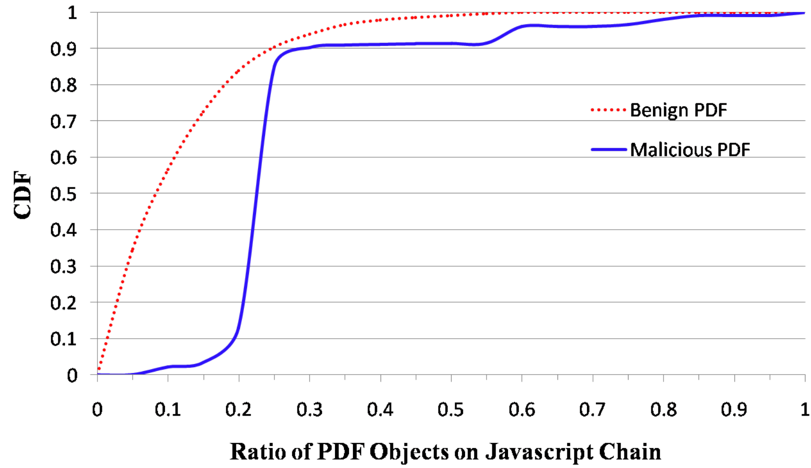


Figure 3.6: Ratio of PDF Objects on Javascript Chain in Malicious and Benign Documents

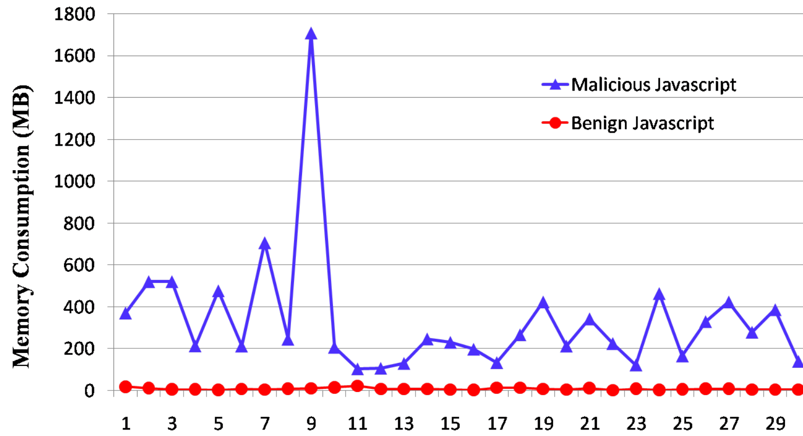


Figure 3.7: Memory Consumption of Malicious and Benign Javascripts

ratio over 0.2. We even found 64 samples with a ratio of 1. This is reasonable since malicious documents usually contain only one blank page. By contrast, the ratio in benign documents presents a quite different pattern. From the dotted line in Figure 3.6, we can clearly see that about 90% of benign documents have a ratio smaller than 0.2 and almost no document has a ratio over 0.6. The results indicate that this feature can effectively distinguish between benign and malicious documents.

The statistical results of the other static features in malicious documents are

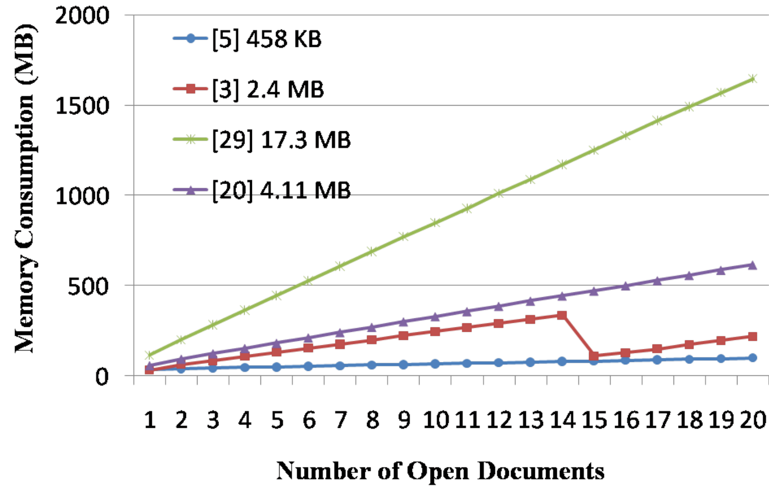


Figure 3.8: Memory Consumption of PDF Reader When Opening Many Documents

shown in Table 3.6. For boolean features, "False" is denoted as 0 and "True" as 1. We found that while empty objects can be found in malicious samples, no benign documents contain empty object. This complies with our intuition that people rarely have incentive to include these junk objects in documents and normally they tend to use automatic tools like `this.addscript()` and [53] to insert Javascript. These tools rarely generate empty objects. Unlike previous two features, more malicious samples use header obfuscation and hex code. As a comparison, we only found three benign documents with header obfuscation and no benign document contains hex code. We believe this is because usually PDF documents are created from other formats like Microsoft Word and LaTeX using automatic conversion tools. Such tools do not obfuscate document header or structure. Finally, only about 1% of malicious samples use multiple levels of encoding, and surprisingly about 3% of them do not use any encoding. In benign documents, we found that all of them use either zero or one level of encoding. Overall, these five features complement with the first feature and enable us to more accurately distinguish between benign and malicious documents.

Memory Consumption: We randomly sampled 30 documents from each of two categories, "Known Benign" and "Known Malicious", respectively. All of the

Table 3.6: Statistics of Static Features of Malicious Documents.

Feature \ Value	0/False	1/True	2	3	6
Header Obfuscation	6792	578	-	-	-
Hex Code	6827	543	-	-	-
Empty Objects	7357	5	4	3	1
Encoding Level	233	7065	40	31	0

30 selected benign documents contain Javascript. Then, we measured the memory consumption of the sampled 60 documents in JS-context and the results are shown in Figure 3.7. As we can see, one malicious sample can consume more than 1700 MB memory. On average, malicious samples consume about 336.4 MB memory while benign documents consume merely 7.1 MB. Moreover, the minimal memory consumed by malicious samples is 103 MB but the maximum by benign samples is only 21 MB. These results indicate that our context-aware monitoring of memory consumption could be an effective feature to differentiate between benign and malicious documents.

Context-aware v.s. Context-free. However, only if the monitoring is conducted in JS-context, will memory consumption be an effective feature. The context-free monitoring could be inaccurate. In order to demonstrate the deficiency of the context-free monitoring, we measure the memory consumption of a PDF reader when different number of documents are opened at the same time. Note that opening many documents simultaneously is a common practice in daily life. In our evaluation, we used Adobe Acrobat 9.0 and four documents with various size from our reference list, including [187] [199] [47] [46]. For each document, we made 20 copies and recorded the memory consumption of Acrobat when different number of copies were opened simultaneously. The results are shown in Figure 3.8. In most cases, the memory consumption increases linearly with the inceasing number of opened documents and it can grow up to 1600 MB. An exception is [187]. When the 15th copy is opened, the memory consumption drops to a lower level and then increases linearly again. We tested many times and this effect appeared in every test. Our speculation is that this specific document triggers

some memory optimization mechanisms in Acrobat. From these results, we can see that it is almost impossible to set an appropriate threshold in the context-free monitoring. A high value could miss a large fraction of malicious documents while a low value may generate many false positives. Besides, as shown in Figure 3.8, the memory increase of [46] is also very large. Thus, in the context-free monitoring, the memory increase of a PDF reader is not a good feature either. By contrast, our context-aware monitoring is much more effective and accurate.

3.4.3 Detection Accuracy

We evaluate the detection accuracy of our prototype, in terms of false positive rate and false negative rate. We tested the malicious samples in VMware Workstation hosting Win XP SP1 with Adobe Acrobat 8.0/9.0 installed. We first describe the parameter configuration of our detector and then present the detection results.

1) Parameter Configuration

First, we normalize non-binary features, including F1, F4, F5, and F8. The normalization rules are listed in Table 3.7. According to Figures 3.6 and 3.7, we set F1 as 1 when the ratio ≥ 0.2 and F9 as 1 when the memory consumption ≥ 100 MB. Similarly, the values of F5 and F6 are set according to Table 3.6. In this way, all 13 features can be represented in binary values.

To set the weights and threshold, we need to meet the criterion that a document is tagged as malicious *iff* at least one JS-context feature and any other features have positive values. The basic idea is that if no suspicious behavior is detected in JS-context, the document contains no malicious Javascript and thus it is out of the scope of our detection. According to the criterion, we set w_1 as 1, w_2 as 9, and the threshold as 10, respectively.

2) Detection Results

We measured the false positive and false negative rates of the tuned detector over all benign documents with Javascript (994) and one thousand randomly selected

Table 3.7: Parameter Configurations in Our System.

Parameter	**Value**
F1	If ratio ≥ 0.2, F1 = 1; else F1 = 0;
F4	If # of empty objects ≥ 1, F4 = 1; else F4 = 0;
F5	If encoding level ≥ 2, F5 = 1; else F5 = 0;
F8	If mem consumption ≥ 100 MB, F8 = 1; else F8 = 0;
w_1	1
w_2	9
Threshold	10

Table 3.8: Detection Results

Category	**Detected Malicious**	**Detected Benign**	**Noise**	**Total**
Benign Samples	0	994	0	994
Malicious Samples	917	25	58	1000

malicious samples. The malicious samples cover vulnerabilities in Javascript interpreter, Flash, U3D (Universal 3D), TIFF and JBIG2 image, etc. The detection results are shown in Table 3.8.

It can be seen that no benign sample is misclassified as malicious, achieving zero false positive. There is only one sample with suspicious behavior in JS-context. However, since there is no other feature with positive value, this sample is still classified as benign. Afterwards, we checked the sample and confirmed that the script uses SOAP for network access. The rest 993 samples are tagged as benign simply because no suspicious JS-context behavior is monitored, although some samples have positive values in other features. Even though Javascript methods like SOAP and ADBC can generate network accesses, we are reluctant to white list them since we cannot decide the maliciousness of the target server.

During the test, 58 ($\sim$6%) of the malicious samples did nothing when opened. Inspecting those samples, we found that these samples exploited either CVE-2009-1492 [52] or CVE-2013-0640 [50] which do not work on Adobe Acrobat 8.0/9.0. As these samples failed to exploit, we excluded them when computing false negative rate. For

Table 3.9: Comparison With Existing Methods

Method	False Positive	True Positive
N-grams [190]	31%	84%
PJScan [137]	16%	85%
PDFRate [194]	2%	99%
Structural [199]	0.05%	99%
MDScan [210]	N/A	89%
Wepawet [41]	N/A	68% [210]
Ours	0	97%

the rest 942 samples, we successfully detected 917, with a detection rate of 97.3%. We examined the 25 undetected samples and we found two reasons that cause the misses. First, although malicious Javascripts in these samples spray the heap, the PDF reader process crashes when the scripts attempt to hijack the control flow. Second, the 25 undetected samples use no obfuscation and thus no static feature contributes to detection. Actually there are more than 25 samples that crash the PDF reader process, but the others are detected by our system via suspicious memory consumption and static features. Although false negatives are unavoidable when malicious PDF fails to exploit, it does not violate our primary goal, i.e., protecting users from damages of malicious PDF.

Table 3.9 compares our method with previous countermeasures in terms of false positive rate and true positive rate. It is clear that our method is comparable with the best fully static methods [194] [199]. Since the malicious samples in our dataset are not the most recent (the latest was captured in Feb. 2013), we cannot fully demonstrate the superiority of our system over the fully static methods. Thus, we further compare our system with other methods by analyzing possible advanced attacks.

- *Our approach v.s. Structural methods*: The mimicry attacks proposed in [154] can effectively bypass these structural methods [194] [199] [155] [137]. However, our approach is immune to the proposed attacks in that we detect the malicious attempts from Javascript rather than how malicious Javascript is stored in PDF.

- *Our approach v.s. Anti-virus Software*: There are a whole bunch of tricks available in the wild to evade anti-virus software [123] [48] [49]. Attackers can easily generate variants using these tricks to defeat anti-virus software. Compared with anti-virus software, our method can effectively detect new variants and zero-day malicious PDF in time because we use the inconcealable system-level behaviors of malicious PDF for detection.

- *Our approach v.s. Dynamic Analysis Tools*: Attackers can subvert existing dynamic analysis tools like CWSandbox [220] using event-triggering and environment-sensitive malcode. Our method does not suffer this limitation since we detect as real users operate on malicious documents.

Based on the analysis of potential advanced attacks, we can see that our method is more robust than existing defense against malicious PDF.

3.4.4 System Performance

To measure the runtime overhead of our method, we run our prototype on 32-bit Windows 7. We performed the tests on a laptop with a 2.53 GHz Intel Core 2 Duo CPU processor and 2 GB of RAM. The performance of each component in our system is presented below.

1) Static Analysis and Instrumentation

Overall, it took about 297.7 seconds to process all 7370 malicious samples, i.e., 0.04 seconds on average for each sample. We also measured the overhead when processing the files with various sizes. We randomly selected three benign and malicious documents, respectively. The sizes of these documents are shown in Table 3.10. One of the malicious samples contains two scripts and the rest of five documents contain only one script.

The execution time of each step in static analysis and instrumentation is shown in Table 3.10. We can see that the overhead is minor for both large and small documents. In particular, it took only about 5.5 seconds to process a 20 MB document.

Table 3.10: Execution Time (in seconds) of Static Analysis & Instrumentation.

PDF Size	Parse & Decompress	Feature Extraction	Instrumentation	Total
2 KB	0.0005	0.0255	0.0183	0.0444
9 KB	0.0008	0.0867	0.0138	0.1014
24 KB	0.0007	0.0726	0.0247	0.0981
325 KB	0.0569	0.0210	0.0236	0.1016
7.0 MB	0.8954	0.4023	0.0773	1.3750
19.7 MB	3.2219	2.0015	0.2761	5.4995

Considering that it could take 20 seconds to download the document (in case of 1 MB/s), the additional delay of 5.5 seconds for processing it is acceptable.

Whereas most of the execution time is spent on feature extraction and instrumentation for small documents, the dominant overhead comes from parsing and decompressing as document size increases, which accounts for over 95% of the total execution time. Besides, for instrumentation, the overhead depends on the number of scripts. That is why it took more time to instrument the 2 KB file than the 9 KB file in Table 3.10. The overhead increase is approximately linear. This is because during feature extraction, we have tagged the PDF objects containing Javascript code and our instrumentation component only needs to locate and instrument them.

In summary, the evaluation results indicate that the component of static analysis and instrumentation incurs minor overhead and can be used for end-host protection.

We also profiled memory overhead. Table 3.11 presents the memory usage during static process. The memory overhead is a little bit high. However, since the front-end component works off-line and the RAM on modern systems can easily accomodate such a memory demand, the overhead is acceptable. Actually, for most documents, the memory overhead of our system is comparable with PDF readers like Adobe Acrobat. In the future work, we will optimize our program and use memory more efficiently.

2) Runtime Detector

The runtime detector with a tiny SOAP server requires about 19 MB memory. Although the detector maintains the state (i.e., all features) for each unknown

Table 3.11: Memory Overhead of Static Analysis & Instrumentation.

PDF Size	# of Python Objects	Memory Consumption
2 KB	74095	5.26 MB
9 KB	74085	5.26 MB
24 KB	74112	5.28 MB
325 KB	74616	5.63 MB
7.0 MB	366845	42.86 MB
19.7 MB	1081771	130.6 MB

open document, we found that the memory usage increases a little as the number of monitored documents increases. Thus, the overhead of our runtime detector is also minor.

We further evaluated the efficiency of our context monitoring code. We manually crafted a set of documents containing various copies of Javascript. The Javascript is from a randomly selected malicious sample. In total, we got 20 documents with 1 to 20 separate scripts in each document. For each crafted document, we measured the total execution time of Javascript before and after instrumentation. When one script is instrumented, the additional execution time incurred by our context monitoring code is about 0.093 seconds. Since most malicious documents in the wild contain only one script, this overhead represents the common case. Note that, although both benign and malicious documents can contain many scripts, in most cases these scripts are invoked sequentially via `/Names` and `/Next`. Thus, only one piece of the context monitoring code is inserted. Basically, the overhead grows linearly as the number of instrumented scripts increases. However, when there are 20 scripts, the overall overhead is still below 2 seconds. Benign documents may contain many singly invoked scripts, but in most cases these scripts are associated with some actions that probably are not triggered simultaneously. Therefore, when the overall overhead is distributed among each script, the performance degradation is still minor. In summary, our context monitoring code is efficient enough for online protection.

3.5 Conclusion

In this dissertation, we develop an effective and efficient hybrid approach—leveraging five novel static features and the context-aware behavior monitoring—for detection and confinement of malicious Javascript in PDF. The static features are designed to detect the obfuscation techniques that are widely used by malicious PDF but usually disregarded by benign documents. We also observe that the indispensable operations for malicious Javascript to compromise target systems rarely occur in JS-context. Based on this observation, we present the static document instrumentation method to facilitate context-aware monitoring of potential infection attempts from malicious Javascript. The intrusive nature of instrumentation method endows our system with immunity to Javascript code and PDF syntax obfuscations. To validate the efficacy of our proposed approach, we conducted a security analysis given an advanced attacker, showing that our method is much more robust than existing defense. The experimental evaluation based on over twenty thousand benign and malicious samples shows that our system can achieve very high detection accuracy with minor overhead.

Chapter 4

TOWARDS AUTOMATED DETECTION OF SHADOWED DOMAINS

4.1 Introduction

The domain name system (DNS) serves as one of the most fundamental Internet components and provides critical naming services for mapping domain names to IP addresses. Unfortunately, it has also been constantly abused by miscreants for illicit online activities. For instance, botnets exploit algorithmically generated domains to circumvent the take-down efforts of authorities [223, 67, 179], and scammers set up phishing websites on domains resembling well-known legitimate ones [118, 202]. In the past, Internet miscreants mostly register new domains to launch attacks. To mitigate the threats, tremendous efforts [74, 66, 121, 204, 113] have been devoted in the last decade to craft real-time blacklists. All of these render it ineffective and inefficient to register new domains for attacks. As a response to the arm race, miscreants have moved forward to more sophisticated and stealthy strategies.

In fact, there is a newly emerging class of attacks adopted by cybercriminals to build their infrastructure for illicit online activities, *domain shadowing*, where instead of registering new domains, miscreants infiltrate the registrant accounts of legitimate domains and spawn subdomains under them for malicious purposes. Domain shadowing is becoming increasingly popular due to its superior ability to evade detection. The shadowed domains naturally inherit the trust of legitimate parent zone and miscreants can even set up authentic HTTPS connections with Let's Encrypt [156]. Even worse, miscreants can virtually create an infinite number of subdomains under a bunch of hijacked legitimate domains and rapidly rotate among them at no cost. This makes it quite challenging to keep blacklists up-to-date and gather useful information for

meaningful analysis. While domain shadowing has been reported in public outlets, like `blogs.cisco.com`, most previous studies only elaborate sporadic cases collected in a short time through manual analysis. It is still unclear how serious the threat is and how to address this domain shadowing problem in a larger scale.

In this dissertation, we present a novel detector to automatically detect shadowed domains and conduct the first comprehensive study of domain shadowing in the wild. This requires addressing several unique challenges however. By design, shadowed domains do not present suspicious registration information and thus all detectors leveraging these data [106, 114, 113] can be easily bypassed. Blindly blacklisting all sibling subdomains of shadowed domains is also infeasible in practice since it can cause large amount of collateral damage. Last but not least, most suspicious DNS patterns identified in previous studies do not work well in domain shadowing. For instance, Kopis [66] analyzes the collective features of all visitors to a domain. However, our study has seen many shadowed domains being visited only once, rendering the collective features insignificant. Such collective features can be applied to malicious apex domains because the domain registration cost can become non-affordable if an apex is used only a few times.

To bootstrap the design of our detector, we collect a set of 26,132 confirmed shadowed domains under 4,862 distinct zones through manually searching and reviewing technical reports by security professionals. Comparing them with legitimate subdomains, we find that the shadowed ones can be characterized and distinguished from two dimensions. On one hand, shadowed domains usually exhibit deviant behaviors and are more isolated from those known-good subdomains under the same parent zone. For instance, most legitimate domains are hosted on reputable servers which usually strictly restrict illicit contents. Due to the nature of their criminal activity and their demand to evade detection and possible take-down, shadowed domains have to host on cheap and cybercriminal-friendly servers. This deviation serves as a prominent indicator of potential shadowed domains. On the other hand, miscreants tend to exploit a bunch of shadowed domains under different parent zones in the same campaign. This

can greatly increase the resilience and stealthiness of their infrastructure. However, such correlation also presents suspicious synchronous characteristics. For instance, shadowed domains in the same campaign usually appear and disappear at the same time.

Based on these observations, we develop a novel system, `Woodpecker`, to automatically detect shadowed domains through inspecting the deviation of subdomains to their parent zones and correlation of shadowed domains among different zones. In particular, we compose 17 features modeling the usage, hosting, activity, and name patterns of subdomains, based on the passive DNS data. Five classifiers (Support Vector Machine, RandomForest, Logistic Regression, Naive Bayes and Neural Network) are then trained using these features. We achieve 98.5% detection rate with about 0.1% false positive rate with a 10-fold cross validation when using RandomForest.

`Woodpecker` is envisioned to be deployed in several scenarios, e.g., domain registrars and upper DNS hierarchy as a complement to Kopis [66], generating more accurate indicators about the ongoing cyber-crimes. In this dissertation, we demonstrate a use case where `Woodpecker` is deployed on an open security service VirusTotal [214]. Specifically, we run our trained classifier over a large scale dataset built using all subdomains submitted to VirusTotal [214] during Feb∼April, 2017 as seeds. The dataset contains 22,481,892 unique subdomains under 2,573,196 parent zones. These domains are hosted on 4,809,728 IP addresses.

Our findings. Applying `Woodpecker` to the daily feeds of VirusTotal, we obtain 287,780 reports, of which 127,561 are confirmed as shadowed domains with a set of heuristics (most of the remaining ones are about malicious apex domains). Our measurement of the characteristics of these shadowed domains indicate that they exhibit quite different properties from conventional malicious domains and thus existing systems can hardly detect the shadowed domains. Our manual assessment of the security measures of domain registrars show that their current practices cannot effectively protect the users. We also observe two interesting cases in our results. First, currently exposed shadowed domains as in the technical blogs are exclusively involved in exploit

kits. However, our detection results show that shadowed domains are also widely exploited in phishing attacks. Another interesting finding is that miscreants also exploit the wildcard DNS records to spawn the shadowed domains.

4.2 Background

We give a brief overview of domain system in the beginning of this section. Then, we describe the schema regarding domain shadowing attack and use one real-world case identified by our detection system to walk through the attack flow.

4.2.1 Basics of Domain Name

Domain name structure. Domain name is presented in the structure of hierarchical tree (e.g. `a.example.com`), with each level (e.g. `example.com`) associated with a DNS zone. For one DNS zone, there is a single manager who oversees the changes of domains within its territory and provides authoritative name-to-address resolution service through DNS server. Top of the domain hierarchy is the root zone, which is usually represented as a dot. So far, the root zone is managed by ICANN and there are 13 root servers operated by 12 organizations. Below the root level is the top-level domain (TLD), a label after the rightmost dot in the domain name. The commonly used TLDs divide into 3 groups, including generic TLDs (gTLDs) like `.com`, country-code TLDs (ccTLDs) like `.uk`, and sponsored TLDs (sTLDs) like `.jobs`. Next to TLD is the second-level domain (2LD) (e.g. `.example.com`), which can be directly registered from registrars (like GoDaddy) if not occupied yet, in most cases. One exception occurs when both ccTLD and gTLD appear in the domain name, like `.co.uk`, and the registrants have to go for 3rd-level domain(3LD), like `example.co.uk`. In this work, we use *effective TLDs (eTLDs)* or *public suffix* to refer to the TLDs directly operated by registrars (like `.com` and `.co.uk`), and *apex domains* (or *apex* in short) to refer to the domains that can be obtained under eTLDs. The registrant owning apex domain is allowed to create *subdomains*, like 3LDs and 4LDs, without asking permission from the

registrar. In the meantime, the registrant takes responsibility of managing the domain resolution, by either running her own DNS server or using other public DNS servers.

DNS record. When a registrant requests a domain name from a registrar, the request is also forwarded to a registry (e.g., Verisign), which controls the domain space under the eTLD and publishes DNS record (or *resource records, (RR)*) in the zone file. Similarly, a subdomain creation request also results in changes in the zone file, except that the request can be handled by the owner herself. A RR is a tuple consisting of 5 fields, <*name*, *TTL*, *class*, *type*, *data*>, where *name* is a fully qualified domain name (FQDN), *TTL* specifies the lifetime in seconds of a cached RR, *class* is rarely used and is almost always `"IN"`, *type* indicates the format of a record, and *data* is the record-specific data, e.g., an IP address of a domain.

Subdomain management. Domain owners can create and manage the subdomains under their apex domains through web GUI or API provided by domain registrars. There are three types of DNS RRs associated with subdomains creation. An `A` record maps a domain name to an IPv4 address, e.g., `foo.example.com A 1.1.1.1`. A `CNAME` record specifies the alias of a canonical domain, e.g., `foo.example.com CNAME bar.another.com`. An `AAAA` record maps a domain name to an IPv6 address, e.g., `foo.example.com AAAA 0:0:0:0:0:0:0:1`. Figure 4.1 shows the web interface of GoDaddy for subdomain creation. Assume the apex is `foo.com` and `Host` field is filled with `shadowed`, a new subdomain `shadowed.foo.com` will be created after the submission of request, which updates the zone file shortly. The domain owner could fill `Host` with `*` to create a *wildcard record*. As a result, any request to the non-existent subdomain (not specified by `A` or `CNAME` record) will be captured and resolved to the corresponding IP.

4.2.2 Domain Shadowing

Web host is a critical asset in the cyber-criminal infrastructure. To prevent the hosts from being easily discovered, like exposing their physical existence from IP, they abuse DNS services and hide the hosts behind the ever-changing domain names. Many

A

Host *	Points to *	TTL *
shadowed	50.63.202.14	1 Hour

Figure 4.1: Adding a subdomain in domain registrar GoDaddy. Assume the apex domain is foo.com. The added subdomain is `shadowed.foo.com`

attackers choose to buy domain names from registrars. Since malicious domains are ephemeral, usually revoked shortly after being detected, they prefer to register many domains with each at low price and short expiration duration. This strategy however leaves the malicious domains more distinguishable from the legitimate domains, when examined by domain reputation systems [113, 74, 152, 106, 153].

Recently, attackers are beginning to compromise the domain system to evade existing detection systems while confining the cost of getting domains. Discovered by Cisco Talos in 2015 [120], Angler, an exploit kit with widespread usage by the underground actors, evolved its infrastructure and used the subdomains under the legitimate domains as redirectors to cover the exploit servers. In particular, the bad actors harvested a large amount of credentials of domain owners (e.g., through phishing email or brute-force guessing) and logged into their accounts to create subdomains. This technique is called *domain shadowing* and such subdomains are called *shadowed domains*.

Domain shadowing is quite effective for the several reasons below. Firstly, many registrants use weak passwords and they never check the domain configuration after creation [95]. In addition, the changes are not submitted to registries' zone file, setting aside the monitoring system of registries. Secondly, there is usually little restrictions over subdomain creation. As long as a domain consists of less than 127 levels and the name length is less than 253 ASCII characters, the domain name is valid. This leaves a virtually infinite space for adversary to rotate domains and evade blacklists. Thirdly, the malicious subdomains inherit the reputation of legitimate apex domains. As information from Whois record poses high impact on the domain score outputted

by many systems [113] and subdomains share the same values as their apex domains, the shadowed domains can easily slip through these systems.

In addition to compromising registrants' credentials, vulnerabilities in registrars and DNS servers could also lead to domain shadowing. For instance, it has been reported that several reputable registrars were breached and massive domain credentials were leaked, including `Name.com` [101], `punto.pe` [178] and `Hover` [207]. As a result, malicious subdomains were able to be created under a large volume of apex domains at the same time. The zone files hosted by the authoritative DNS servers were targets of domain hackers, who manipulate the RR data to change or add domains [129].

Scope. Specifically, our threat model assumes that attackers have gained administrative control of legitimate 2LD and attempt to spawn new subdomains for illicit purposes. In particular, we do not detect modifications and deletions of existing subdomains because they either cause denial of service which is out of our scope or can be easily spotted by domain owners. Moreover, our study focuses on domain shadowing that lurks in legitimate 2LD. While subdomains under malicious 2LD might also be classified as shadowed, we rely on existing tools like PREDATOR [113] to filter out the malicious 2LD. Finally, we address domain shadowing campaigns exploiting legitimate 2LD in bulk and do not handle targeted attacks on a single 2LD.

In this work, we aim to detect shadowed domains *created* by domain hackers *in bulk*. While the existing research revealed that such technique was mainly used by exploit kit (see the description of our ground-truth data in Table 4.2), we consider all attacks leveraging this technique, like phishing, into our study. Changing and deleting subdomains without owners' consent, which could achieve the same goal or cause service interruption, are not considered in this dissertation, given that they are more likely to be observed and used less frequently. While subdomains could be created under malicious apex domains, they are not the focus of our study, and existing tools gauging domain reputation like PREDATOR [113] could be leveraged here. Targeted attacks like APT (Advanced Persistent Threat) try to operate on a small number of domains, including subdomains under legitimate apex domains. Detecting targeted

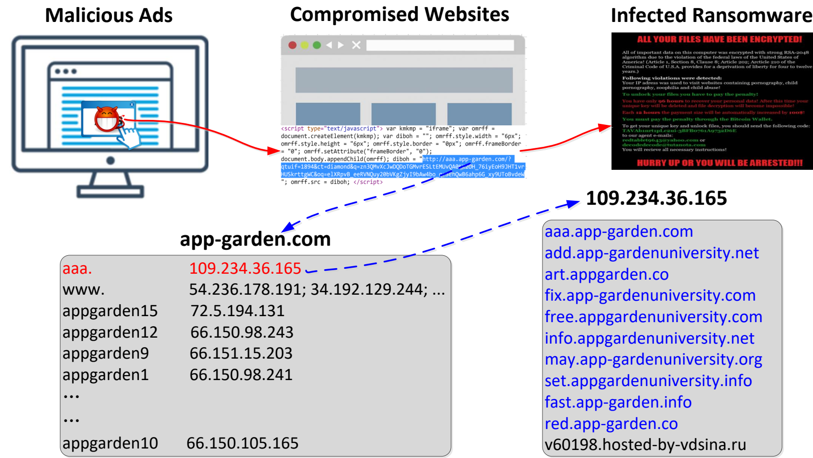


Figure 4.2: Shadowed domains used in a campaign of EITest Rig EK in April, 2017. `app-garden.com` is a legitimate apex domain.

attack automatically is still a paramount challenge for security community [103], due to its nominal signal overwhelmed by huge amount of data. We do not expect individual subdomains in these cases can be effectively detected by our system and leave the research as a future direction.

4.2.3 Real-world Example

Here we demonstrate how domain shadowing powers attackers' operations using a real-world case recently discovered by our system (illustrated in Figure 4.2). We found the shadowed domains showed up in the passive DNS data (our dataset is described in Section 4.3.2) and later documented in a security website [157] . One such domain is `aaa.app-garden.com`, created under a legitimate 2LD `app-garden.com` and redirecting users' traffic from compromised doorway sites to *Rig Exploit Kit (EK)* [198], within a malware distribution campaign called *EITest*. In particular, the doorway sites served malicious advertisements created by attackers and the JavaScript code redirected the visitor to a sequence of compromised sites, till arriving at `aaa.app-garden.com`, which kept Rig EK's drive-by-download code. If the malicious code executed successfully in user's browser, a ransomeware will be downloaded and encrypted victim's files.

By inspecting the data relevant to the shadowed domains, we discovered several unique features about such attack. The shadowed domain aaa.app-garden.com pointed to an IP address that is quite different from the apex app-garden.com and other sibling subdomains, like www.app-garden.com. More specifically, the shadowed domain was associated with an IP in Russia while all other subdomains were linked to IPs in US. Looking into the domains linked to the 109.234.36.165 (in total 10 from our data), we found 9 of them share the similar apex names as app-garden.com (e.g., app-garden.co). Notably, all 9 apex domains were registered by Cook Consulting, Inc, with one in April 2011, six in May 2014 and two in March 2017 [1]. We speculate that the domain hacker obtained the login credential and injected subdomain into many apex domains under victim's account. Also interesting is that meaningful single word, like info and free, were used to construct the malicious subdomains. As such, detectors looking for random domain names, like DGA detector [223, 67], would be evaded at high chance.

4.3 Automatic Detection of Shadowed Domains

In response to the emerging threat of domain shadowing, in this section we present our design of an automated detection system, Woodpecker. We first overview its workflow and deployment scenarios. Then, we describe the dataset used for training and testing. Finally, we elaborate on the features we use to distinguish shadowed and legitimate domains.

4.3.1 Overview

We could follow conventional approaches, like content or URL analysis, to detect shadowed domains. However, after our initial exploration, we found that these

[1] The app-garden.com was registered through domain proxy and the registrant information is not available through Whois query. However, the domain was registered at the same time as one of the domains.

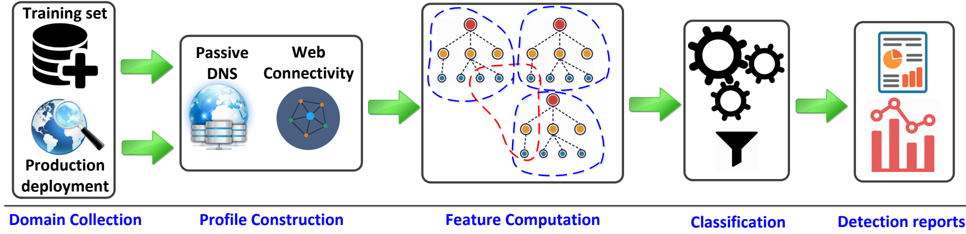


Figure 4.3: Workflow of `Woodpecker`.

approaches are not suitable. Many shadowed domains are used as redirectors. Finding the gateways, e.g., compromised sites, is a non-trivial task. Even if we are able to find the shadowed domains and download the content, we may still fail to classify them correctly when they only serve seemingly benign redirection code. Compared to domains owned by attackers, the registration information of a shadowed domain is identical to that of the benign apex domain, which undermines the effectiveness of many approaches based on domain registration.

Users' visits to shadowed domains would be observed by DNS servers and further collected by a passive DNS (PDNS) database. Erasing the traces from DNS servers and PDNS is considerably more difficult than compromising websites and domain accounts. As such, we decide to analyze the DNS data to solve our problem. Though the information underlying the DNS data is much more scarce than web content, it is still sufficient to distinguish shadowed and legitimate domains, due to two key insights. First, shadowed domains serve a different purpose from the legitimate parent domains and sibling subdomains: for instance, they could be associated with IPs far from their parents' and siblings', leading to prominent *deviation*. Second, to make malicious infrastructures resilient to take-down efforts, attackers prefer to play domain fluxing and rotate shadowed domains. In the meantime, the IPs covered by them are limited, leading to abnormal *correlation*, especially when they are under apex domains whose owners have no business relations.

Our detection system, `Woodpecker`, is driven by those two insights and runs a

novel deviation and correlation analysis on the PDNS data. It takes three steps to detect shadowed domains. Given a set of subdomains $\mathbb{S}$ observed at a certain vantage point (e.g., enterprise proxy and scanning service), we first build the profiles for each apex of $\mathbb{S}$ using the data retrieved from the PDNS source. Assume an apex **D** is represented by a set of tuples:

$$\mathbf{D} = \{\ s_i \mid s_i :=< name_i, rrtype, rdata, t_f, t_l, count > \}$$

where $name_i$ is the FQND under D, `rrtype` and `rdata` represent the type (e.g., `A` record) and data (e.g., IP) fields within the answers returned by DNS servers , t_f and t_l denote the time when an individual `rdata` is first and last seen, and `count` is the number of DNS queries that receive the `rdata` in response.

In the second step, `Woodpecker` aggregates these profiles and characterizes the subdomains using a set of 17 significant features from the dimensions of deviation and correlation. In addition to the data from PDNS, we also query a public repository of web crawl data to measure the connectivity of domains (only extracting web links). Finally, a machine-learning classifier is trained over a labeled dataset and is further applied to large unlabeled datasets to detect shadowed domains. Figure 6.2 depicts the workflow of `Woodpecker`.

Deployment. `Woodpecker` is a lightweight detector against shadowed domains, which only requires passive DNS and publicly crawled data. We envision `Woodpecker` to be deployed in several scenarios. It can help domain registrars like GoDaddy to detect domains whose subdomains are added in an unexpected way, and hence allows them to notify domain owners promptly. The operators of DNS servers can deploy our system to trace and mitigate Internet threats. The administrators of organizational networks can use the output of our system to amend their blocked lists (i.e., whether to block a subdomain or an apex domain). Finally, it can be deployed by public scanning services, like VirusTotal [214], to analyze submitted URLs/domains and provide more accurate labels. When these services are used as blacklists and a site is blocked, knowing the label is essential for the owner to diagnose the root cause [75].

Dataset	Category	# of Domains	# of Apex	Farsight		360	
				# of Domains	# of IP	# of Domains	# of IP
$D_{shadowed}$	Shadowed	26,132	4,862	21,958	1,188	7,121	965
$D_{unknown}$	Unlabeled siblings of $D_{shadowed}$	-	-	34,586	27,630	8,573	10,609
D_{pop}	Legitimate popular	-	8,719	8,965,818	3,596,441	1,081,112	645,763
D_{nonpop}	Legitimate unpopular	-	2,500	713,154	349,874	80,920	61,507
D_{vt}	Daily feeds from VirusTotal	-	2,573,196	-	-	22,481,892	4,809,728

Table 4.1: Training and test datasets. Columns 3∼4 include all domains we manually collected and thus some cells like those of $D_{unknown}$ do not have data. Columns 5∼8 present the number of domains obtained from two PDNS, Farsight and 360, respectively.

Source	Campaign	# Indicators
blogs.cisco.com	Angler [120, 64]	16,580
blog.talosintelligence.com	Neutrino [205], Angler [197, 208], Sundown [82]	9,536
heimdalsecurity.com	Angler [107]	5
blog.malwarebytes.com	Neutrino [193], Angler [57, 211]	9
proofpoint.com	Angler [189]	2
	Total	26,132

Table 4.2: Sources of confirmed domain shadowing.

4.3.2 Dataset

To bootstrap our study, we collected domains from different sources and queried the PDNS data to build profiles. Below we describe how these data were collected and summarize them in Table 4.1.

Shadowed domains. Obtaining a list of shadowed domains requires a lot of manual effort. While there are many public blacklists documenting malicious domains, we have not found any such list for shadowed domains specifically. Hence, we rely on web search[2] (using keywords like "domain shadowing" and "shadowed domain") to find all relevant articles. After manually reviewing that information, the indicators (i.e., malicious domains/IPs/hashes) in the articles are downloaded. The subdomains hosted under known malicious apex domains and directly under third-party hosting services are removed for dataset sanitization. Overall, we managed to collect 26,132 known shadowed domains under 4,862 apex domains, as listed in Table 4.1[3]. Table 4.2 summarizes this dataset, and we name it $D_{shadowed}$. While all shadowed domains in

[2] Searching Google and `otx.alienvault.com`, a platform sharing threat intelligence.

[3] We rely on a list documenting the public suffix in domain names to extract the apex [162].

$D_{shadowed}$ are used for exploit kits, we are able to discover other types of usage, like phishing, from the testing dataset (elaborated in §4.5.1).

Legitimate domains. We collected legitimate domains as another source to train the classifier. The data comes from two channels. First, we chose domains that are consistently ranked among the top 20,000 from 2014 to 2017 by Alexa [160], and we obtained 8,719 2LDs in total. These popular domains usually have many subdomains that cover a broad spectrum of services, including web, mail, and file downloading. Solely relying on popular domains can introduce bias to our system, so we also obtained non-popular legitimate domains from a one-week DNS trace collected from a campus network. The DNS trace was anonymized and desensitized for our usage. We scanned these domains using VirusTotal and excluded all of the malicious ones (alarmed by at least one participating blacklist). Further, we randomly sampled 2,500 2LDs that were ranked below 500,000 by Alexa in 2017. The two datasets are denoted as D_{pop} and D_{nonpop}. The volume of subdomains found from our legitimate datasets is not very extensive due to the rate limit placed by the PDNS provider, as described later.

VT daily feeds. We evaluated the trained model based on the data downloaded from VT, as a showcase to demonstrate that `Woodpecker` can be readily integrated into security services. In particular, we queried for a live feed of reports on all URLs submitted to VT during February∼April 2017 on a daily basis. For each submitted subdomain s_i, we queried VT to obtain the domain report and IP report to include additional information for later result validation. All subdomains without IP and apex information were filtered out, in order to reduce unnecessary queries submitted to PDNS. We further excluded subdomains one level under web hosting services and dynamic DNS based on the `category` field from the VT domain report (e.g., "web hosting" and "dynamic DNS"). This dataset is denoted as D_{vt}, which contains 22,481,892 unique subdomains under 2,573,196 apex domains.

Passive DNS data. We queried the PDNS data of two security companies, Farsight Security [100] and 360 Security [176], to obtain aggregated DNS statistics for apex

```
## From 360
{"rrname": "eu.account.amazon.com", "rrtype": "A",
"rdata": "52.94.216.25;", "count": 31188,
"time_first": 1477960509, "time_last": 1494290720}
## From Farsight
{"rrname":"aws.amazon.com.","rrtype":"A",
"rdata":["54.240.255.207"], "count":63,
"time_first":1302981660,"time_last":1318508315}
```

Figure 4.4: Two sample records for subdomains under `Amazon.com` from 360 and Farsight (field explanation is covered in Section 4.3.1).

domains in all datasets (we used a wildcard query, like `*.example.com`, to retrieve the data associated with all subdomains of `example.com`), except D_{VT}. We did not query Farsight for D_{VT} due to its daily rate limit. Our account granted by 360 does not have such restrictions and we queried 360 for all apex domains in D_{VT}. Figure 4.4 shows two sample records from 360 and Farsight.

The columns 6∼8 in Table 4.1 present the obtained data. As shown, different PDNS databases have varying coverage. The evaluation of the impact of different PDNS sources is presented in §4.4.3. For $D_{shadowed}$, their siblings under the same apex domains might be added by attackers but missed by security companies. It is desirable to determine whether `Woodpecker` can detect new shadowed domains among them. As such, we constructed another dataset $D_{unknown}$, which includes all unlabeled siblings of $D_{shadowed}$.

4.3.3 Features of Domain Shadowing

`Woodpecker` inspects the PDNS data collected from the global sensor array to detect shadowed domains. Prior to our work, there have been several approaches using PDNS data to detect malicious domains in general, like Notos [65], Exposure [74], and Kopis [66]. However, these systems are not good choices for finding shadowed domains, due to their different features (e.g., ephemeral and readable names) and appearance in many different attack vectors (not only those used by botnet).

By examining the ground-truth set $D_{shadowed}$, we found a set of features unique to shadowed domains, which are essentially divided into two dimensions.

- Deviation from legitimate domains under the same apex. How subdomains are created and used differs greatly between legitimate site owners and domain hackers. To name a few, legitimate subdomains tend to be hosted close to the apex, while shadowed domains are hosted by bullet-proof servers with much fewer restrictions whose IP is far from the apex. A site owner usually creates subdomains gradually while shadowed domains are added in bulk around the same time. The homepage of the apex domain (or `www` subdomain) usually contains a link to legitimate subdomains while shadowed domains are isolated, since the registrar and apex website run different systems and compromising them at the same time is much more difficult.

- Correlation among shadowed domains under a different apex. Inspecting a single apex is not always effective. On the other hand, shadowed domains under a different apex might be correlated, when an attacker compromises multiple domain accounts and uses all injected subdomains for the same campaign. For instance, shadowed domains under a different apex might be visited around the same time and point to the same IP address, which rarely happens for legitimate subdomains under a different apex.

In the end, we discovered 17 key features for the detection purpose, under four categories: usage, hosting, activity, and name, as listed in Table 4.3. All features related to deviation can be defined as $D(s_i, \mathbb{S}_{apex(s_i)})$, where $\mathbb{S}_{apex(s_i)}$ represents all known-good domains under the same apex of s_i. Labeling all known-good domains is impractical when processing massive amounts of data. Instead, we simply consider the apex domain and `www` subdomain as known-good. Site owners usually create `www` subdomains for serving web content after the domain is purchased, so they are rarely taken by attackers. The correlation features are extracted from subdomains hosted together, i.e., sharing the same IP. We choose IP to model correlation since legitimate websites tend to avoid sharing the IP with attackers. Below we elaborate the details

Category	Feature ID	Feature Name	Dimension	Novel
Subdomain Usage	F1	Days between 1st non-www and apex domain	D	✓
	F2	Ratio of popular subdomains under the same apex domain	D	✓
	F3	Ratio of popular subdomains co-hosted on the same IP	C	✓
	F4	Web connectivity of a subdomains	D	✓
	F5	Web connectivity of subdomains under the same apex domains	D	✓
	F6	Web connectivity of subdomains co-hosted on the same IP	C	✓
Subdomain Hosting	F7	Deviation of a subdomain's hosting IPs	D	✓
	F8	Average IP deviation of subdomains co-hosted on the same IP	C	✓
	F9	Correlation ratio in terms of co-hosting subdomain number	C	[74]
	F10	Correlation ratio in terms of co-hosting apex number	C	[74]
Subdomain Activity	F11	Distribution of first seen date	C	✓
	F12	Distribution of resolution counts among subdomains on the same IP	C	✓
	F13	Reciprocal median of resolution counts among subdomains on the same IP	C	✓
	F14	Distribution of active days among subdomains on the same IP	C	✓
	F15	Reciprocal median of active days among subdomains on the same IP	C	✓
Subdomain Name	F16	Diversity of domain levels	C	✓
	F17	Subdomain name length	C	[67, 113]

Table 4.3: Features used in our approach to detect shadowed domains. Feature dimensions **D** and **C** denote Deviation and Correlation, respectively. Although some features use the same data source as previous work, e.g., resolution counts as in [141, 60], we model them in different ways.

of each feature.

4.3.3.1 Subdomain Usage

This category characterizes how subdomains are visited, their popularity and web connectivity.

Days between first non-`www` and apex domain. We check when the first non-`www` subdomain was created under the apex. We found that many compromised apex domains only run websites, whose only legitimate subdomain is a `www` domain. Therefore, a new subdomain created suddenly should be considered suspicious. Assume `Date(d)` is the date when a domain `d` is first seen. We compute this feature as $F1 = \frac{1}{\log(Date(s) - Date(apex(s)) + 1)}$, where `s` is the first non-`www` subdomain under its apex and `apex(s)` denotes its apex. If there are no subdomains or all subdomains are created on the same day as their apex, this feature is set to 1.

Ratio of popular subdomains. Miscreants usually generate names of their shadowed domains algorithmically. We observe that the names tend to avoid being overlapped with popular subdomain names, as changing the existing subdomain is not among the attacker's goals. Based on this observation, we define two features, the ratio of popular

www	mail	remote	blog	webmail
server	ns1	ns2	smtp	secure
vpn	m	shop	ftp	mail2
test	portal	ns	ww1	host
support	dev	web	bbs	ww42
mx	email	cloud	1	mail1
2	forum	owa	www2	gw
admin	store	mx1	cdn	api
exchange	app	gov	2tty	vps
govyty	hgfgdf	news	1rer	lkjkui

Table 4.4: List of top 50 popular subdomain names.

subdomains under the upper apex and on an IP[4]. Specifically, given a suspicious subdomain s, we compute $F2 = \frac{|\{POP(d_i)\}|}{|\{d_i|2LD(d_i)==2LD(s)\}|}$ and $F3 = \min_{j=1..n}\{\frac{|\{POP(d_i)\}|}{|\{d_i|IP(d_i)==IP_j(s)\}|}\}$, where IP_j is the jth IP of s. For $POP(d_i)$, we only consider subdomains with only one more level than their apex. For example, `www.foo.com` is a popular subdomain under `foo.com` while
`www.a.foo.com` is not. We examined the Forward DNS names collected by Project Sonar [99] and selected the top 50 names for popular subdomains, as listed in Table 4.4.

Web connectivity. Shadowed domains are irrelevant to the services provided by their apex, sibling and hosting servers. As a result, they are not connected to the homepage or other subdomains through web links, while connections between legitimate subdomains and apex are more likely established. Furthermore, a shadowed domain is hardly accessible to web crawlers that aim to index web pages, and cloaking is frequently performed.

Here we use the data collected by public web crawlers, including Internet Archive [68] and CommonCrawl [85], to measure the connectivity[5]. For each subdomain s, we issue a query to Internet Archive and CommonCrawl. If any page under s is found to be indexed, this feature, denoted as `F4 = WEB(s)`, is set to 1. Otherwise, it is set to 0.

---

[4] We issue additional PDNS queries to obtain subdomains not shown in the collected datasets for an uncovered IP.

[5] We did not query search engines like Google, because queries are blocked when sending too many.

Additionally, we compute $F5 = \frac{\sum WEB(d_i)}{|\{d_i|2LD(d_i)==2LD(s)\}|}$ and $F6 = \min_{j=1..n}\{\frac{\sum WEB(d_i)}{|\{d_i|IP(d_i)==IP_j(s)\}|}\}$, or the ratio of reachable subdomains under the same apex and same IP. Although accurately assessing connectivity is impossible, we observe that these two crawlers have good coverage of the legitimate domains and hence provide a solid approximation.

4.3.3.2 Subdomain Hosting

Deviation of hosting IP. Shadowed domains are usually hosted on IP addresses distant from their apex domain and other known-good subdomains. By contrast, legitimate subdomains tend to be hosted within one region, e.g., within the same autonomous system (AS). Given an apex domain $A = \{< f_i, l_i, ip_i >\}_{i=1..n}$ and its subdomains $S = \{< f_i, l_i, ip_i >\}_{i=1..m}$, where f_i and l_i denote the first and last seen date of ip_i, the deviation (F7) is computed as,

$$Dev(A, S) = max_{j=1..m}\{min_{i=1..n}\{\psi(A_i, S_j)|A(f_i) < S(f_j)\}\} \quad (4.1)$$

where $\psi(A_i, S_j)$ is a function that computes the deviation score between two IP records. It is defined as,

$$\psi(A_i, S_j) = \sum_{C\in\{IP,ASN,CC\}} w_k(C[A_i] \neq C[S_j]) \quad (4.2)$$

where w_k is the weighted penalty for the binary difference between A_i and S_j in IP, AS number (ASN), and country code (CC). We empirically set the weights to 0.3, 0.2, and 0.5. For example, if A_i and S_j share the same IP, the deviation score is 0 (ASN and CC are identical, too). Otherwise, if A_i and S_j share the same ASN but not the same IP, the deviation score will be 0.3. If all of these attributes are different, the deviation score reaches 1.0. Additionally, we compute the average deviation ($F8$) of all subdomains hosted on the same IP.

Correlation ratio. In order to characterize the co-hosting properties of subdomains, we define two features. First, given a subdomain $s = \{IP_j\}_{j=1..n}$, we compute how many subdomains are co-hosted with s, specifically $F9 = \min_{j=1..n}\{\frac{1}{\log(|\{d_i|IP_j(d_i)==IP_j(s)\}|+1)}\}$. This feature alone cannot distinguish shadowed and legitimate subdomains, as we found that some IPs are hosting tens of thousands of legitimate subdomains, probably used by CDN. To address this issue, we count the distinct apex whose subdomains are hosted together with s. The reason behind using this feature is that most site owners prefer to have a dedicated host with a dedicated IP after we filter out the domains that belong to shared hosting and dynamic DNS. We compute $F10 = \min_{j=1..n}\{\frac{1}{\log(|\{2LD(d_i)|IP_j(d_i)==IP_j(s)\}|+1)}\}$ for this feature.

Take the case described in §4.2.3 as an example to explain how the feature values are computed. There are 11 subdomains from 11 distinct 2LDs co-hosted with `aaa.app-garden.com`. Therefore, the two feature values are $(\frac{1}{\log 12}, \frac{1}{\log 12})$. By contrast, legitimate subdomains under `app-garden.com`, like `appgarden15.app-garden.com`, do not co-host with any other subdomains, and their feature values are $(\frac{1}{\log 2}, \frac{1}{\log 2})$.

4.3.3.3 Subdomain Activity

To evade blacklists, miscreants tend to create many shadowed domains under different hijacked apex domains, using and discarding them simultaneously, which results in strong but abnormal correlation. However, the legitimate subdomains are more independent from one another. In this study, we measure the correlation from three aspects: first seen date, resolution count, and active days.

Our goal here is to determine how consistent these features are across different subdomains. To this end, we convert each feature into a frequency histogram and compare it to a crafted histogram when all subdomains share the same value, and then use Jeffrey divergence [45] to measure their difference. Specifically, given a set of values V, we first count the weighted frequency of each value, resulting in a set

$W = \{< w_i, \frac{w_i}{|V|} >\}_{i=1..n}$. We then derive a new set W' by setting $< w_i, 1 >$ if w_i has the largest frequency $\frac{w_i}{|V|}$; otherwise $< w_i, 0 >$. Finally, Jeffrey divergence is computed over W and W'.

Distribution of first seen date. Given a subdomain s, we compute the Jeffrey divergence of the first seen date (in the format of MM-DD-YYYY) among all subdomains hosted together with s. This feature is denoted as $F11$.

Resolution count. The visits to shadowed domains tend to be more uniform, as they are rotated in regular intervals. The visits to legitimate domains are much more diverse, and certain subdomains like `www` usually receive substantially more visits. Also, legitimate domains tend to receive more visits than malicious ones. To model this property, we define two features, Jeffrey divergence ($F12$) and the reciprocal of median ($F13$) of resolution count.

When computing this feature, we aggregate all the resolution counts associated with the observed IPs for a domain name. Therefore, even if the mapping between an IP address and a domain name is not one-to-one, e.g., when IP-fluxing is played by attackers, the resolution count is not diluted. On the other hand, when an IP is shared across different domain names, e.g., when domain-fluxing is abused, this feature is not affected either, because resolution counts are separated between individual domain names, regardless of their IPs.

Note that while a malicious apex domain is oftentimes mapped to multiple IPs (IP-fluxing), the attackers we studied here usually use subdomains in a thrown-away manner because it costs them nothing to create. More specifically, we observe that a shadowed domain is normally used only for a very short period of time (most of them less than five resolutions) and mapped to one IP.

Active days. The feature above may raise alarm when legitimate subdomains are rarely visited. As a complementary method, we also compute the active days of subdomains, or how long a subdomain and IP pair is witnessed. This works particularly well

when an attacker frequently changes the hosting IP. By contrast, IPs for legitimate domains are more stable, resulting in longer active days. Similar to the resolution count, we use two features, Jeffrey divergence ($F14$) and the reciprocal of median ($F15$) of active days.

4.3.3.4 Subdomain Name

Similar to DGA domains [223, 67, 179], many shadowed domains are algorithmically generated, instead of being manually named. We model the name similarity of all co-hosted subdomains under two numerical features. Note that the randomness of characters (e.g., entropy of words) within one domain name is not considered by us, because we found many shadowed domain do have meaningful label, like `info`.

Diversity of domain name levels. Shadowed domains belonging to the same campaign are usually generated using the same template, and thus their domain levels are the same. However, legitimate domains hosted on the same IP have less uniform domain levels. Similar to the above features, we compute the Jeffrey divergence ($F16$) for all of the subdomains hosted together.

Subdomain name length. For this feature, we remove the substring matching the apex from each subdomain and compare the remaining length. When subdomains in the same group have different levels, we pad them to the maximum level by adding empty strings. Assume the prefix of subdomain is $\mathbb{N} = \{< n_i >_{i=1..m}\}$, where n_i is the ith level, we compute the Jeffrey divergence for each level of name, denoted as Jeffrey($\mathbb{N}_i$), and then take the mean value, as $F17 = \frac{\sum_{i=1}^{m} Jeffrey(N_i)}{m}$.

4.4 Evaluation

In this section, we present the evaluation results of `Woodpecker` on labeled datasets described in §4.3.2. We first compare the overall performance of five different

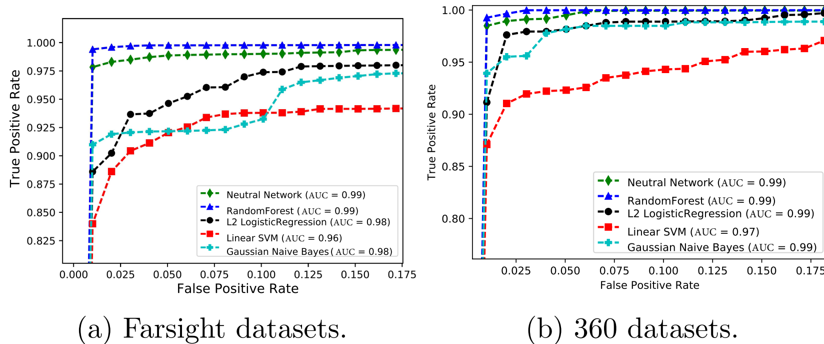
classifiers on three ground-truth datasets. Then, we analyze the importance of each feature. Finally, we evaluate `Woodpecker` on two testing sets, $D_{unknown}$ and D_{vt}.

4.4.1 Training and Testing Classifiers

We first test the effectiveness of our detector over the ground-truth datasets, $D_{shadowed}$, D_{pop}, and D_{nonpop} through the standard 10-fold cross-validation. We partition the data based on the apex domains to ensure that for each round of testing, we have subdomains to test from the apex domains unseen in the training phase. Specifically, subdomains in $\frac{9}{10}$ of the randomly selected apex domains fill the training set, and those in the remaining $\frac{1}{10}$ apex domains fill the testing set.

We use the scikit-learn machine-learning library to prototype our classifiers [186]. We compare five mostly used machine-learning classification algorithms, including RandomForest, SVM with a linear kernel, Gaussian Naive Bayes, L2-regularized Logistic Regression, and Neutral Network. Figure 6.12 illustrates the receiver operating characteristic (ROC) curves of these classifiers, when using Farsight and 360 PDNS to build domain profiles. The x-axis shows the false-positive rate (FPR), which is defined as $\frac{N_{FP}}{N_{FP}+N_{TN}}$, and the y-axis shows the true-positive rate (TPR), which is defined as $\frac{N_{TP}}{N_{TP}+N_{FN}}$. We observe that all classifiers can achieve promising accuracy on both PDNS data sources. To reach a 90% detection rate, the maximum FPR is always less than 3% for all classifiers, suggesting that `Woodpecker` can effectively detect shadowed domains.

Evidently, RandomForest outperforms the other classifiers in all cases. This is mainly because domain shadowing detection is a non-linear classification task. Thus, RandomForest and Neutral Network consistently outperform Logistic Regression and linear SVM. Meanwhile, our dataset is not very clean, e.g., shadowed domains being falsely labeled as benign for training. RandomForest can handle noisy datasets very well [78]. Moreover, some features that `Woodpecker` extracts could be inaccurate, e.g., the resolution count and active days. These features depend on the vantage points



(a) Farsight datasets. (b) 360 datasets.

Figure 4.5: Performance comparison of classifiers under 10-fold cross-validation. The number of trees used in RandomForest is 100. All other classifiers use the default configuration in scikit-learn.

where DNS queries are monitored. RandomForest is more robust to those errors [78]. Finally, RandomForest can effectively handle imbalanced training datasets [78].

Next, we draw more details on the false positives and negatives. We focus on the best performing classifier RandomForest only and use it for all follow-up experiments. Due to the space limit, we only present the results on Farsight data (results on 360 have a similar distribution) in the rest of evaluations. In total, `Woodpecker` misclassifies 222 shadowed domains as legitimate (false negatives) and six legitimate ones as shadowed (false positives). We manually inspect these instances to understand the cause of the misclassification. First, about one third of these shadowed domains have snapshots in `Archive.org`. Nevertheless, most of these snapshots were captured several years ago. By contrast, most of the legitimate subdomains in our dataset have much fresher snapshots. For example, the last snapshot of `extranet.melia.com` dated back to 2008, but the subdomain was used for an attack in 2015. We speculate that these subdomains have been abandoned by domain owners (i.e., no longer serving any web content) but were later revived by attackers for illicit purposes. One approach to address this inaccuracy is to set an expiration date for snapshots. Second, the majority of the missed shadowed domains co-host either with siblings only or with a few other subdomains, which lessens the effectiveness of our correlation analysis. On the other hand, the features of all six false positives resemble shadowed domains. For instance,

Rank	Feature	Score	rank	Feature	Score
1	F10	0.26188	10	F8*	0.03374
2	F2*	0.13213	11	F12*	0.03183
3	F7*	0.11509	12	F16*	0.03128
4	F17	0.06493	13	F3*	0.02852
5	F5*	0.0623	14	F15	0.02395
6	F9	0.05221	15	F13*	0.02309
7	F1*	0.04496	16	F6*	0.01491
8	F14	0.04424	17	F4*	0.00036
9	F11*	0.03451			

Table 4.5: Importance of features. Features marked with an asterisk (*) are novel.

they are all hosted in countries different from their apex domains, and all subdomains on the same IP are visited only a few times.

4.4.2 Feature Analysis

We assess the importance of our features through a standard metric in the RandomForest model, namely mean decrease impurity (MDI) [79], which is defined as,

$$MDI(X_m) = \frac{1}{N_T} \sum_{T} \sum_{t \in T:(s_t)=X_m} p(t) \Delta f(s_t, t) \tag{4.3}$$

where X_m is a feature, N_T is the number of trees, p(t) is the proportion of samples reaching node t (N_t/N), $v(s_t)$ is the variable used in split s_t, and $f(s_t, t)$ is an impurity decrease measure (Gini index in our case). Table 4.5 shows the score of each feature with the novel ones marked with an asterisk. As we can see, three of the top five features are novel, suggesting that using known features is not sufficient to capture shadowed domains.

We further evaluate the impact of different groups of features. Figure 4.6 compares the performance of `Woodpecker` when deviation-only and correlation-only features are used. Interestingly, `Woodpecker` can still achieve a 95% TPR with less than 0.1% FPR when only features in deviation dimension are used. As such, the operators behind `Woodpecker` can choose to trade a little accuracy for higher efficiency, since computing correlation features are more resource-consuming.

In addition, we assess the performance of features under each of the four categories. The results are shown in Figure 4.7. Except for the feature of subdomain

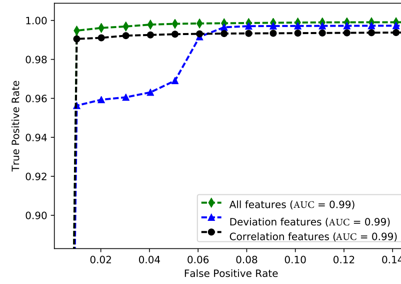


Figure 4.6: ROC of RandomForest on Farsight data when all, deviation-only (F1, F2, F4, F5, F7) and correlation-only (all others) features are used.

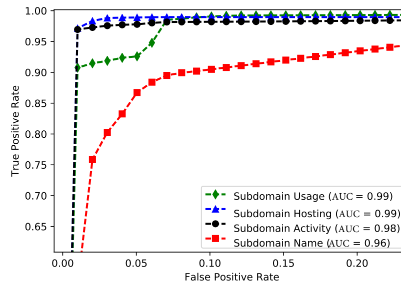
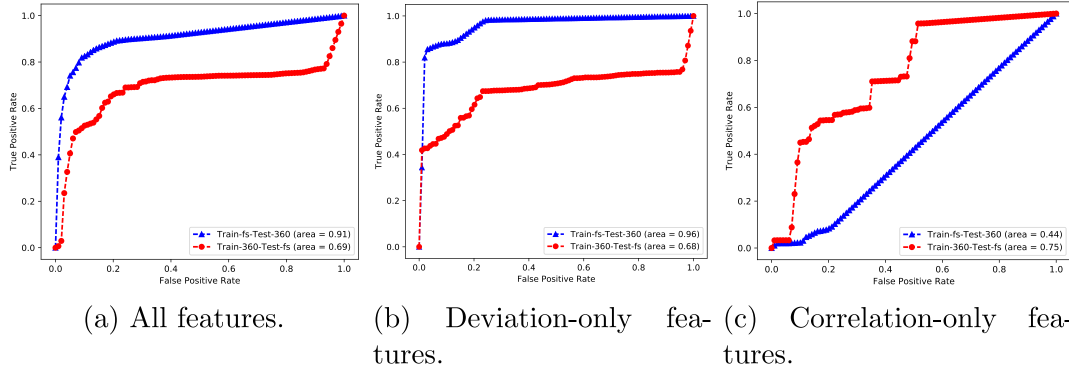


Figure 4.7: ROC of RandomForest on Farsight data when features in a single category are used.

name, all other feature categories produce a reasonable performance. The feature of subdomain name does not perform well because many legitimate services like cloud platforms and content delivery network (CDN) also have seemingly algorithmically generated domain names.

In summary, according to our analysis, it is almost impossible for attackers to evade `Woodpecker` by manipulating a few features. Instead, they would need to manipulate many features in both deviation and correlation dimensions, and the cost is non-negligible. Take the feature of hosting IP deviation as an example. We observe that most compromised apex domains use their registrars' hosting services. GoDaddy is particularly popular as it is also the largest domain registrar. In order to confuse this feature, attackers can either change the IP of an apex domain, which will be discovered by site owners immediately, or host their shadowed domains on GoDaddy



(a) All features. (b) Deviation-only features. (c) Correlation-only features.

Figure 4.8: Performance of `Woodpecker` using RandomForest trained and tested on different PDNS sources. FS stands for Farsight.

as well. However, unlike less reputable and bullet-proof hosting services, GoDaddy is a poor choice for attackers, due to its much more stringent policies and actions against malicious content.

4.4.3 Generality of Trained Models

The training and testing stages of our last experiments are carried out on an identical dataset. We want to confirm whether `Woodpecker` can be trained on one dataset and then applied to another dataset, and how its performance is impacted. To this end, we evaluate two configurations, i.e., training the model on Farsight and testing on 360, and vice versa. We exclude all subdomains in Farsight that overlap with the 360 dataset, and thus the training and testing datasets have no overlap.

Figure 4.8 illustrates the results when different dimensions of features are used. We find that both configurations cannot produce comparable results to our prior settings when all features are used, which might indicate that `Woodpecker` needs to be re-trained when being deployed on different vantage points. We further examine the performance when deviation-only features are used. Interestingly, the result of the model trained on Farsight is significantly improved, while the result of the model on 360 remains almost the same. Moreover, the performance of both models decreases significantly when correlation-only features are used. The plausible reason behind this

is the uneven coverage of PDNS sources, which greatly impacts the correlation analysis. For instance, given an IP address, Farsight may observe tens of subdomains hosted on the IP, while 360 might observe only one or two. Hence, a model trained on Farsight could derive totally different feature weights compared to 360.

In summary, when deviation-only features are used, `Woodpecker` can be migrated among different vantage points without re-training. A model trained on a PDNS source would yield better results when tested on the same source.

4.4.4 Evaluation on $D_{unknown}$

We now evaluate `Woodpecker` on $D_{unknown}$ to examine whether we can accurately distinguish legitimate and unknown shadowed subdomains under known hijacked apex domains.

Among the 34,586 unknown subdomains in $D_{unknown}$ (Table 4.1), `Woodpecker` reports 10,905 shadowed domains. Since this dataset is unlabeled, we have to validate the result through manual investigation. We use a set of rules, after confirming their validity with an analyst from a security company. In particular, we consider a subdomain as a true positive (1) if it has been deleted from the authoritative DNS servers, (2) if it is hosted together with those in $D_{shadowed}$, (3) if its name follows the same pattern as known shadowed ones, (4) if it is reported by other security companies, and (5) if it is not running any legitimate business. After these steps, we confirm 10,866 as true positives and 39 as false detections. The false detection rate is thus 0.35%, which is consistent with our results on $D_{shadowed}$. Measuring FNR is very challenging, given there are still over 20K subdomains remaining. Here we randomly sample 50 apex domains in $D_{shadowed}$ and examine all the subdomains. In the end, we do not find any new shadowed domains missed by `Woodpecker`.

4.4.5 Evaluation on D_{vt}

Finally, we apply `Woodpecker` to a large unlabeled dataset, D_{vt} built from the daily feeds of VT, consisting of more than 20M subdomains that are recorded by 360.

This dataset is more representative in that it covers many types of malicious domains, either shadowed or non-shadowed. Many legitimate subdomains are also contained in this dataset. As demonstrated in §4.4.3, `Woodpecker` achieves its best performance when it is trained and tested using data from the same PDNS source. Therefore, we use `Woodpecker` with RandomForest that is trained on 360 data for this evaluation. In total, `Woodpecker` reports 287,780 shadowed domains (1.28% of the total subdomains) under 23,495 apex domains.

Given these results, we first sanitize them by removing subdomains under malicious apex domains, since our main goal is to detect malicious subdomains created under legitimate apex domains. Then, we verify whether the remaining subdomains are indeed shadowed. Such a validation process is very time-consuming and challenging. The best way is to report all of them to domain owners and registrars and wait for their responses. However, previous studies [143] have shown that most are unresponsive. Even finding all of the recipients is impossible in short term. So, we take a best-effort approach instead and categorize these domains based on clustering and manual analysis. In the end, they can be labeled into five categories.

Expired apex domains. First, we examine the Whois of all apex domains and find that 1,782 out of the 23,495 apex domains have already expired, which account for 45,093 of the reported subdomains. We exclude all subdomains under these expired apex domains, because there is no sufficient information left to us to determine the legitimacy of the apex. This rule may remove some true positives: We check the apex in $D_{shadowed}$ and find that about 18% have expired. As a future improvement, we could run `Woodpecker` more promptly when the data is downloaded from our vantage point.

Lead fraud [138, 147]. Second, we observe that 341 of the *in-use* apex domains covering 86,886 reported subdomains are involved in lead fraud, a type of online scam that solicits user's personal information. They are identified by scanning domain names using a set of keywords attributed to known lead-fraud campaigns, like `rewards`. One such example is `oiyzz.exclusiverewards.6053.ws`. Manual sampling over these domains (and apex) shows most of them are indeed carrying out lead fraud. We check

the features of these domains and find that they show similar patterns to domain shadowing. For instance, their subdomains are hosted in different ASes and sometimes in different countries from their apex domains.

Deleted subdomains. After expired and lead fraud domains are excluded, we further run DNS probing over the remaining 155,801 subdomains to see whether they are resolvable. It turns out that 29,565 had already been deleted. We consider these domains very suspicious as their injected DNS records might be purified by domain owners, especially when in most cases their siblings are still resolvable.

Heuristics based pruning. We further validate the remaining resolvable domains using three heuristics. First, we construct the prefix patterns based on known-shadowed domains, which are rarely used by legitimate subdomains, like `add.` and `see.`. Second, we search for the subdomains alarmed by at least one vendor in VT but whose apex domains have no alarms. Third, we cluster all subdomains based on their IP addresses. If one subdomain in a cluster has been confirmed in previous steps, we consider all others to be confirmed as well. In this way, we successfully identify 97,996 additional shadowed domains.

Manual review. Finally, we manually review the remaining 28,240 subdomains. In order to make this task tractable, we cluster these subdomains based on their apex domains and analyze the top 100 large clusters and 200 other random apex domains. We observe that 98 apex domains (covering 14,090 subdomains) are quite suspicious in that we cannot find any information about the hosting sites from Google search results. Meanwhile, many of them have been reported by security companies. Among them, 41 are potential DGA (Domain Generation Algorithm) domains, which we speculate are registered by attackers. In the remaining set, 868 subdomains come from eight dynamic DNS and three CDN services like `dyn-dns.org` and `Limelight CDN`, and they are labeled as false positives. In addition, 358 are falsely alarmed as they run the apex owner's legitimate business, e.g., `live.bilibili.com`, totaling 1,226 false positives. We are unable to confirm the remaining 12,924 subdomains due to their

sheer volume.

Summary. In total, 127,561 shadowed domains are confirmed under 21,228 apex domains, hosted on 4,158 IP addresses. Compared to $D_{shadowed}$, only 254 subdomains under 216 apex domains are overlapped. Note that our validation and sanitization of the data is best-effort: True shadowed domains could be eliminated, and legitimate subdomains might be included. We would like to emphasize two lessons learned during this validation process. First, dynamic DNS and CDN services are the main sources of false positives reported by `Woodpecker`. Therefore, to improve accuracy, we have built whitelists for dynamic DNS and CDN services [184, 98]. Second, subdomains under malicious apex domains could exhibit similar features to shadowed domains and trigger alarms. To distinguish them, blacklists focusing on apex domains like VirusTotal and other domain reputation systems [113] can be leveraged. The whitelists and blacklists can be incorporated into `Woodpecker` to further improve its accuracy.

4.5 Measurement and Discoveries

`Woodpecker` identifies in total 127,561 shadowed domains from various sources, which significantly surpasses the community's knowledge about this attack vector (only 26,132 shadowed domains were reported before our study). This sheer amount of data offers us a good opportunity to gain a deeper understanding of this issue. We conduct a comprehensive measurement study on the collected data and report our findings below.

We first count the number of compromised apex domains, and show the trend in Figure 4.9. When there are many shadowed domains under an apex domain, we use the year of its first observed shadowed domain. The earliest case that we observed happened in 2014. Since then, the number of affected apex domains increases substantially every year. Because our dataset only contains data before May 2017, we observe fewer shadowed domains in 2017. This result indicates that domain shadowing is becoming increasingly rampant and deserves more attention from the security community. Next, we conduct in-depth analysis from three aspects.

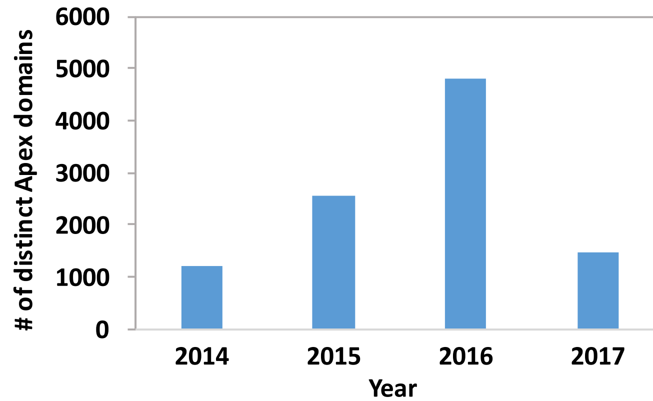


Figure 4.9: Trend of domain shadowing.

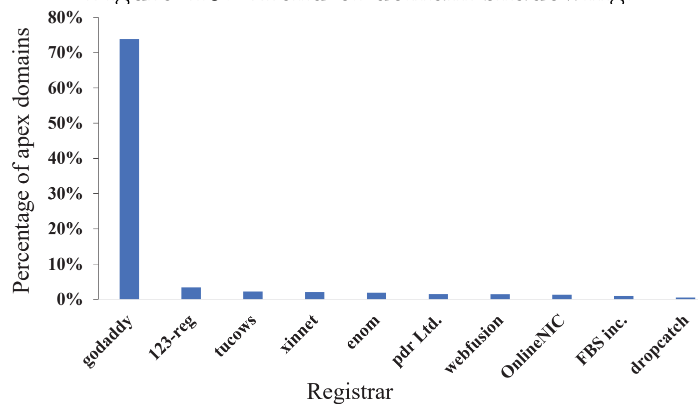


Figure 4.10: Top 10 registrars in terms of distinct apex domains with shadowed domains.

Affected registrars. In total, the shadowed domains trace back to 117 registrars. Figure 4.10 shows the top 10 registrars in terms of distinct apex domains. We can see that GoDaddy accounts for more than 70% of compromised apex domains while the percentage for other registrars is much lower. Considering that GoDaddy shares about 32% of the domain market, which is much greater than the second largest one (6%), this result does show that domain shadowing is a serious issue for GoDaddy, but this does not necessarily indicate that it is the most vulnerable registrar. There are also small registrars gaining high rankings in our result. The registrant buying domains under them should check their account settings more cautiously.

To assess how these registrars protect their users, we manually examine the

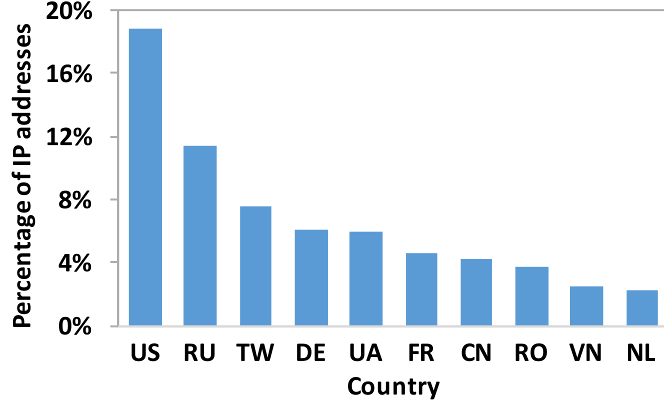


Figure 4.11: Distribution of IPs in the top 10 countries.

security measures of the top 5 registrars. Table 4.6 shows their password requirements for registrants, whether they enforce two-factor authentication (2FA), and how they notify owners about modifications. We observe that 2FA is either not provided or disabled by default. This situation is alarming and disappointing, as the best account defense does not play a role here. Also, no registrars notify users when the DNS records are modified in the default settings.

Registrar	Password Length	2FA	Notification of Modifications
GoDaddy	>9 chars with 1 capital, 1 lower and 1 digit	SMS	No
123-reg	>9 chars with 1 capital, 1 digit and 1 special	No	No
Tucows†	-	-	-
XinNet	8-16 with 1 digit	Yes	No
eNom	6-20 with 1 number and 1 special	SMS	No

Table 4.6: Security policy of the top 5 registrars in our detection. †Tucows is the owner of eNom and Hover etc. and provides services under them.

Hosting IP. In total, 4,158 IP addresses associated with shadowed domains spreading in 91 countries are discovered. Figure 4.11 illustrates the top 10 countries and their percentages. As shown, most of these IPs are located in the United States (US) and Russia (RU). We further find that the IPs in US and RU are widely spread as they belong to 161 and 137 ASes, respectively. This indicates that domain shadowing is used for many different campaigns or by different attackers. We check these IP addresses in VirusTotal and find that 1,499 IPs were not alarmed. Therefore, malware-evidence

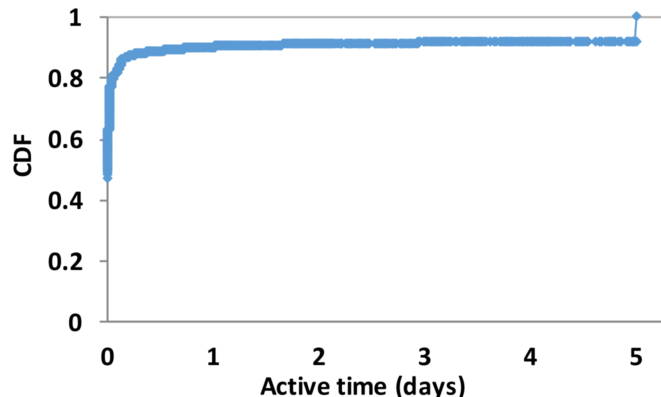


Figure 4.12: CDF of the active days of shadowed domains.

or blacklist-based features used in Notos [65] and Kopis [66] will not work well for our settings.

Shadowed domains. Finally, we analyze the characteristics of shadowed domains and their apex. Basically, the number of shadowed domains under an apex is quite random, from one up to 2,989 with the average number at six. Most shadowed domains have a short lifetime and are mostly (85%) resolved for less than five times per IP. Figure 4.12 shows the CDF of the active days of shadowed domains. Among them, 85% are observed for only one day. This indicates that miscreants rotate shadowed domains quickly, in a similar fashion as fast-flux networks [119].

Previous work [74] uses the TTL value to identify malicious domains. We do not use it for our problem, since it is usually not distinctive on the ground-truth set. We verify this design choice on the entire set by sending DNS queries for 10,000 randomly sampled resolvable shadowed domains. The result confirms our prior observation that the value is either the same as their apex or within the normal range of other legitimate domains.

By cross-checking with VT, we find that 126,384 shadowed domains were submitted to VirusTotal but only 14,134 subdomains were alarmed. In other words, security companies have not yet devised and deployed an effective solution, and we believe that `Woodpecker` can provide great value in tackling domain shadowing.

4.5.1 Case Studies

There are two new findings uncovered in our measurement study. First, in addition to serving exploit kits, shadowed domains are also used for other attack vectors like phishing. Second, wildcard DNS records are also leveraged to create shadowed domains.

Phishing. All currently reported shadowed domains like those in Table 4.2 are exclusively involved in exploit kits. However, `Woodpecker` identifies many phishing attempts that exploit shadowed domains. One ongoing campaign is `paypal.com.webapps.`*random-characters*`.`
`5degreesfalmouth.co.uk`. We consider the apex domain as legitimate because we find that its Facebook account is actively maintained[6] and it is advertised on reputable websites[7]. There are many similar cases like `verifychase.com-u.mescacompany.com` and `apple.com.`*random-characters*`.yclscholarship.org`.

However, we did not see a phishing site impersonating compromised apex domains. We assume this is probably because most compromised apex domains are not popular enough, and only a limited number of victims can be targeted.

Wildcard DNS records. While an arbitrary number of subdomains can be spawned by inserting many `A` and `CNAME` records, the simplest way to create many records is to exploit wildcard DNS records. One prominent advantage of using wildcard records is that attackers do not need to use templates or algorithms to generate subdomain names. However, it is at the cost that the prone to be spotted by domain owners. `Woodpecker` identifies many shadowed domains spawned by wildcard records[8], like `bookstore.hyon.com.cn` and `blackhole.yilaiyin.com`. We determine these cases to be true domain shadowing by incorporating several pieces of evidence. First, most of these apex domains are proven to be legitimate based on the information collected

[6] https://www.facebook.com/5DegreesWest/

[7] https://www.falmouth.co.uk/eatanddrink/5-degrees-west/

[8] Wildcard record is identified if the record `*.apex.com` can be resolved.

through Google search. Second, all wildcard records under these apex domains point to IP `180.178.59.74`, and several other domains hosted on the IP have at least one alarm in VirusTotal. Our detected subdomains have no alarms because they were never submitted to VirusTotal. Finally, VirusTotal reports that two malware samples communicated with this IP. We observe that all of these apex domains are registered from the same registrar, XinNet. Considering that there have been several data breaches against this registrar [129, 200] in the past, we speculate that these apex domains are probably victims in these incidents.

4.6 Discussion

`Woodpecker` is designed to detect subdomains *created in bulk* by attackers. The malicious subdomains falling out of this category might be missed, like modification of existing subdomains or the subdomains created under malicious apex domains, as elaborated in §4.2.2.

An attacker who knows the features used by `Woodpecker` could change her strategy for evasion. To hinder the effectiveness of our correlation features, the attacker can choose to cut off the connections between the shadowed domains, like spreading them to larger pool of IPs. However, this change would increase the attacker's operational cost. Alternatively, the servers linked to the shadowed domains can be co-hosted with other benign servers on the same set of IPs in order to confuse our detector. So far, we find such co-hosting rarely happened, since many shadowed domains are related to the core components of malicious infrastructures, like exploit servers, which are preferably hosted by bullet-proof providers [109]. In addition, placing the services on reputable hosting providers increases their risk of being captured. To evade our deviation analysis, the attacker can learn how the legitimate services on the apex domains are managed and then configure the shadowed domains to resemble her target. For instance, increasing the observed days until reaching the same level of the apex domain is likely effective against `Woodpecker`. However, such changes are more noticeable to site owners. To summarize, evading `Woodpecker` requires meticulous adjustment from

the side of adversary, while the side-effects are inevitable (e.g., raising operational costs and awareness from site owners).

When the subdomains under malicious apex domains exhibit similar features to shadowed domains, they may be detected by `Woodpecker` as well. We believe capturing such instances is also meaningful, especially for security companies. Meanwhile, tools focusing on malicious apex domains, like PREDATOR [113], can be used here for better triaging.

To some extent, the effectiveness of `Woodpecker` depends on the training data. While some previous works rely on data not directly accessible to the public [65, 74, 66], we want to highlight that all of our data is obtained from sources open to researchers and practitioners. Thus, deploying our approach is considerably easier. So far, `Woodpecker` runs in a batch mode, i.e., when PDNS data from a large amount of domains and IPs are available. For real-time detection, `Woodpecker` can be configured to load all existing domain/IP profiles into memory and run the trained model whenever there is an update.

4.7 Conclusion

In this dissertation, we present the first study on domain shadowing, an emerging strategy adopted by miscreants to build their attack infrastructures. Our study stems from a set of manually confirmed shadowed domains. We find that domain shadowing can be uniquely characterized by analyzing the deviation of subdomains from their apex domains and the correlation among subdomains under different apex domains. Based on these observations, a set of novel features are identified and used to build our domain shadowing detector, `Woodpecker`. Our evaluation on labeled datasets show that among five popular machine-learning algorithms, Random Forest works best, achieving a 98.5% detection rate with an approximately 0.1% false positive rate. By applying `Woodpecker` to the daily feeds of VirusTotal collected in two months, we can detect thousands of new domain shadowing campaigns. Our results are quite alarming and indicate that domain shadowing has become increasingly rampant since 2014. We also

reveal for the first time that domain shadowing is not only involved in exploit kits but also in phishing attacks. Another prominent finding is that some miscreants do not use algorithmically generated subdomains but exploit wildcard DNS records.

Chapter 5

ALL YOUR DNS RECORDS POINT TO US: UNDERSTANDING THE SECURITY THREATS OF DANGLING DNS RECORDS

5.1 Introduction

As one of the most critical components of the Internet, the Domain Name System (DNS) provides not only vital naming services but also fundamental trust anchors for accessing Internet services. Therefore, it has always been an attractive target to attackers [71], [125], [128]. In order to ensure the authenticity and integrity of DNS systems, tremendous efforts have been devoted to protecting both client and server mechanisms [84], [91], [177], [216]. In particular, a suite of security mechanisms like DNSSEC [69] have been deployed to secure the communication channels between DNS servers and clients. However, little attention has been paid to authenticating the links between DNS servers and those resources to which DNS records point.

New Threat. In this dissertation, we investigate a largely overlooked threat in DNS: a dangling DNS record (Dare), which could be easily exploited for domain hijacking due to the lack of authenticity checking of the resolved resources. A DNS record, represented in a tuple `<name, TTL, class, type, data>`, is essentially a pointer, where the `data` field points to the machine that hosts the resources for the `name` field. Similar to pointers in a program, a DNS record can also become dangling. When a service accessed by the `name` field discontinues, the domain owner will release the machine to which the `data` field points and should also purge the related DNS records. Unfortunately, in practice, domain owners often forget to do the cleaning, thus resulting in dangling DNS records. Conventional wisdom holds that Dare is by and large safe. To better understand this threat, we conduct the first comprehensive study on

exploitable Dares (*unsafe Dares*) in the wild. In particular, our work reveals that Dare is a real, prevalent threat.

We initiate our study by scrutinizing the DNS specifications, during which four types of security-sensitive Dares are identified, including Dare-A, Dare-CN, Dare-MX, and Dare-NS. To exploit unsafe Dares, an adversary needs to gain control of the resources in the `data` fields of DNS records. There are two types of resources in Dares: IP addresses and domain names. We present three attack vectors that an adversary can harness to hijack these resources. (1) In the first attack vector, we observe that cloud platforms have become a popular choice for modern websites. In clouds, physical resources, especially the public IP address pool, are shared among all customers. Unfortunately, in practice, many domain administrators mistakenly trust these ephemeral and publicly allocable resources, potentially generating all types of Dares. In a sense, this attack vector is probability-based since the IP allocation in clouds is generally random. (2) Modern websites extensively use third-party services. To integrate a third-party service into a website, a domain owner needs to add an `A` or `CNAME` record in the authoritative DNS (aDNS) servers and claim the ownership of the (sub)domain on the owner's third-party service account. Any service account that successfully claims ownership of a (sub)domain can control the content of that (sub)domain. Surprisingly, most third-party services do not verify such a claim, implying that an adversary can potentially claim and control any (sub)domain that has been abandoned by its original owner. The second attack vector is thus to hunt for the Dares linked to abandoned third-party services. (3) Since a domain can expire [141], the third attack vector is simply to search for the expired domains in the `data` fields of DNS records.

Large-scale measurement study. Given the three attack vectors, we then assess the magnitude of the unsafe Dares in the wild. We conduct a large-scale measurement on four datasets, one containing the apex domains in the Alexa top 1 million spanning seven years and the other three containing the subdomains in the Alexa top 10,000, 2,700 `edu`, and 1,700 `gov` zones, respectively. For the first attack vector, we develop a simple tool called IPScouter to automatically milk IP addresses in the clouds,

especially the two largest clouds, Amazon EC2 [2] and Microsoft Azure [30]. Due to its probabilistic nature, IPScouter cannot enumerate the whole IP address space. We therefore assess the magnitude of *potential dares* by filtering all live IP addresses. For the second attack vector, nine of the most popular third-party services are measured. For the third attack vector, we crosscheck the `WHOIS` data and the domain registrars to identify these expired domains.

In total, 791 confirmed and 5,982 potential Dares are successfully found in our measurement study. Especially, Dares exist in all four datasets, indicating a widespread threat. Even more worrisome, Dares can be found in 335 high-valued zones, including those in `edu`, `gov`, and Alexa top 10,000. By exploiting these Dares, an adversary can significantly enhance many forms of fraud activities (e.g., spamming, spear phishing, and cookie hijacking). With the emergence of automated and free Certificate Authorities (CA) like `Let's Encrypt` [24], adversaries can even have the hacked subdomains signed and set up a "genuine" HTTPS website.

Mitigations. We posit that the fundamental cause of unsafe Dares is *the lack of authenticity checking* of the ephemeral resources to which DNS records point. We thus propose three mechanisms that DNS servers and third-party services can adopt to mitigate unsafe Dares. (1) We first design a mechanism that allows aDNS servers to authenticate these machines to which `A` records point. (2) In the case of third-party services, we propose breaking the resolution chain of the dangling `CNAME` records by adopting a safer isolated name space for each user of a service. (3) Finally, we advocate that aDNS servers should periodically check the expiration of domains to which DNS records point.

5.2 DNS Overview

The structure of DNS is organized as a hierarchical tree, which is shown in Figure 5.1. The second- and sometimes third-level domains are registered by enterprises or end-users for connecting their local computing resources to the Internet. Any enterprise/user can own a domain name if it has not yet been registered by another

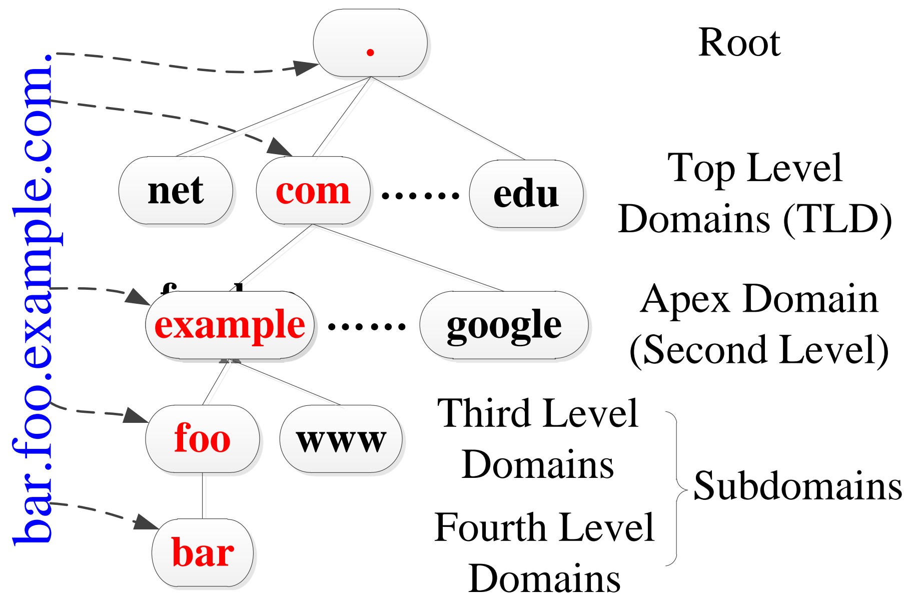


Figure 5.1: The hierarchy of DNS.

user. The further levels of domains are usually called subdomains, typically used to designate a particular host, like web and mail servers.

The conversion between a domain name and an IP address is called DNS resolution. Figure 5.2 illustrates the workflow of DNS resolution when a client visits www.foo.com for the first time. The stub resolver on the client queries a recursive DNS (rDNS) server that can be either local or remote, i.e., outside the local network (❶). In the case of a cache miss, rDNS will initiate queries recursively to the root server, the .com Top Level Domain (TLD) server, and the authoritative DNS (aDNS) server of foo.com (❷ ∼ ❻). Finally, the authoritative server of foo.com will respond with the corresponding IP address of www.foo.com (❼ ∼ ❽). Once the client obtains the IP address, it can connect to the website hosting server (❾ ∼ ❿).

Figure 5.3 shows sample records on the .com TLD server and the aDNS server for the example described in Figure 5.2. Each line of the DNS data represents a resource record (RR), which is a five-tuple data structure <name, TTL, class, type, data>. The fields <name, class, type> serve as the key to data, and TTL is the time-to-live

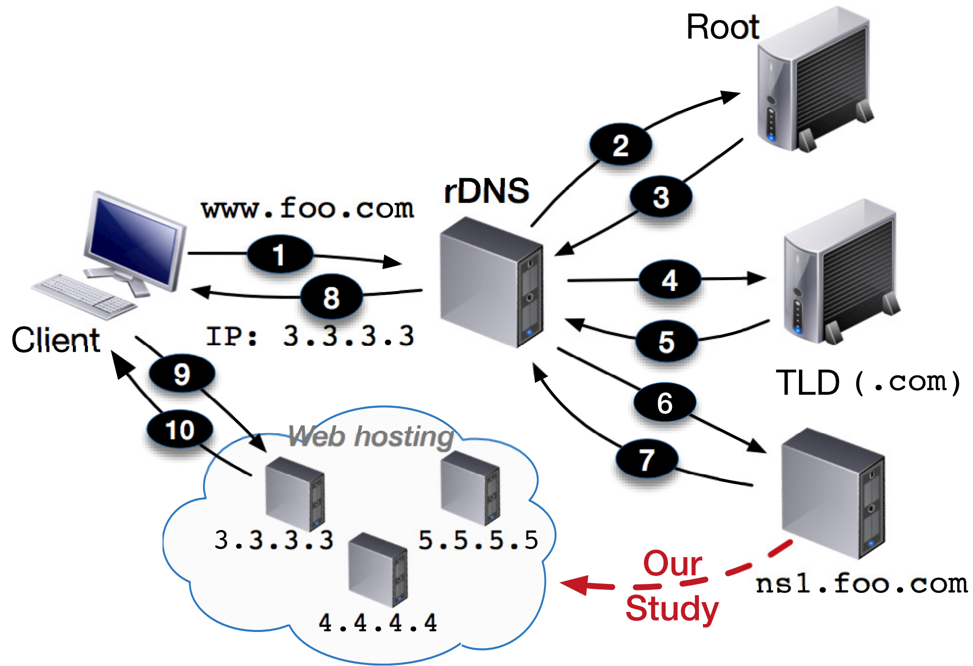


Figure 5.2: The workflow of DNS resolution for `www.foo.com`.

in seconds that determines the lifetime of cached DNS records.

```
;; sample portion of .com zone file.

    foo.com. NS ns1.foo.com.
    foo.com. NS ns2.foo.com.

    ns1.foo.com. A 1.1.1.1
    ns2.foo.com. A 2.2.2.2
```

```
;; sample records from ns1.foo.com

   bar.exmaple.com. CNAME www.foo.com.
   www.foo.com. A 3.3.3.3
```

Figure 5.3: Sample portion of a TLD zone file and DNS records on a resolver. For brevity, the TTL and class fields are omitted.

5.3 Dangling DNS Records

Our work is inspired by the use-after-free vulnerabilities that exploit the dangling pointers in software. The data field of a DNS record is essentially a pointer, as exemplified in Figure 5.4. In this example, the data field, `1.2.3.4`, points to the machine that hosts the content of `www.foo.com`. Later, when the subdomain is no longer

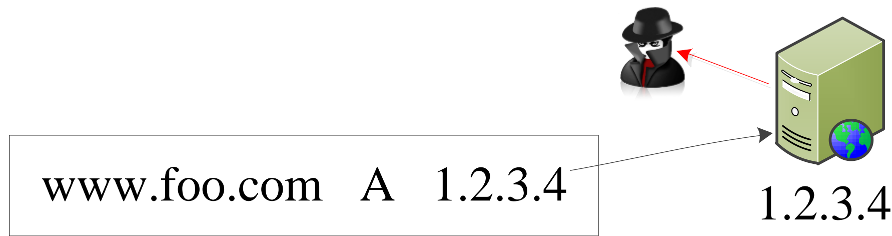


Figure 5.4: An example of a dangling A record.

needed, the domain owner will release the IP address. The corresponding DNS record becomes dangling if the domain owner forgets to remove it from the authoritative DNS server. In general, we define a dangling DNS record as:

Dangling DNS Record (Dare). A DNS record `r:=<name, TTL, class, type, data>` is dangling if the resource to which the `data` field points is released.

Currently, there are more than 40 types of DNS RRs. After scrutinizing the semantics of each type of DNS RR, we identify four security-sensitive records if they become dangling. These records are listed in Table 5.1. Obviously, not all Dares are vulnerable to be exploited. For example, given a Dare-A in Figure 5.4, if an adversary cannot easily obtain the IP `1.2.3.4`, this Dare-A is safe. Here we further define unsafe Dares.

Unsafe Dare. A Dare is unsafe if the abandoned resource could be manipulated by a third party other than the one who controls the `name` field.

In the following, we first review some key details of the DNS records in Table 5.1 and then present three approaches that attackers can harness to exploit the unsafe Dares.

5.3.1 Security Sensitive Dares

Dare-A. An `A` record maps a domain name to an IPv4 address. All requests to the `name` field of an `A` record will be directed to and handled by the host at the

Dare	RR	Description
Dare-A†	A	Returns an IPv4 address
Dare-CN‡	CNAME	Alias of a name to another
Dare-MX	MX	Maps to a list of message transfer agents
Dare-NS	NS	Delegate to an authoritative name server

Table 5.1: Types of security-sensitive dangling DNS records. †Our work currently covers IPv4 only. ‡DNAME is semantically similar to CNAME, so we do not consider DNAME separately.

IP address. Thus, the domain name will be compromised if the IP address could be acquired by a third party other than the original domain name owner.

Dare-CN. A `CNAME` record specifies that a domain name is an alias for another domain name, the "canonical" domain name. For instance, `www.foo.com` in Figure 5.3 is the canonical domain name of its alias, `bar.example.com`. A request to the alias will be resolved to its canonical domain name, which is further resolved to an `A` record. Note that exploiting Dare-CN has almost the same effect as exploiting Dare-A.

Dare-MX. An `MX` record specifies the mail server responsible for accepting emails on behalf of the domain. In the case of multiple MX records, users can set a priority to each one and the server with the lowest value (i.e., highest priority) will be used first. In the following example, an email client will contact `a.mail.com` and `b.mail.com` first (usually in a round-robin manner); if both fail to respond, `c.mail.com` will then be contacted. Note that an `MX` record is not necessary to receive emails. When no `MX` is used, the `A` record of the domain (e.g., `foo.com`) will be treated as an implicit `MX` [21]. If a Dare-MX could be exploited, an adversary may be able to send and receive emails in this vulnerable domain.

```
foo.com. 60 MX 10 a.mail.com.
foo.com. 60 MX 10 b.mail.com.
foo.com. 60 MX 20 c.mail.com.
```

Dare-NS. An `NS` record delegates a domain to an aDNS server for answering queries about names under that domain. There also exists an `A` record to provide the IP address for the aDNS server, which is dubbed as a glue record. Normally, there are multiple `NS` records serving a single domain, and the resolvers need to choose one

aDNS server for further querying. The aDNS server selection [226] can be (1) hitting the first server, (2) randomly selecting one, or (3) sorting the records based on a local-defined rule like RTTs. To force DNS resolvers to use a Dare-NS, attackers can leverage several techniques like Denial-of-Service attacks and NS pinning [115]. If a Dare-NS could be exploited, adversaries will set up a malicious aDNS and direct visitors to any IP address. Due to the transitive trust in DNS [183], the impact of Dare-NS is amplified to all those domains that directly or indirectly depend on it.

5.3.2 IP in Cloud

Every Dare in Table 5.1 is finally resolved to an IP address and thus adversaries can directly obtain the IP address to exploit unsafe Dares. For instance, if adversaries can obtain `1.2.3.4` in Figure 5.4, all subsequent requests to `www.foo.com` will then be handled by adversaries. Whether an IP address is obtainable highly depends on how a domain is hosted.

Figure 5.5 illustrates the three typical paradigms of modern domain hosting. In the first case, a domain is hosted on a dedicated machine with an IP allocated from the address blocks owned by the domain owner. Many large organizations like universities adopt this paradigm for most of their domains. However, the majority cannot afford dedicated hosting, and they usually host their domains using third-party services like GoDaddy [18]. In the normal configuration of these third-party services, many domains are hosted on a single server sharing the same IP address. A user only owns and controls the allocated storage space on the server. In both paradigms, a Dare-A is generally safe because adversaries cannot easily obtain the IP address to which the Dare-A points.

However, nowadays, more and more domains are migrated to the clouds. In particular, a customer can potentially obtain any public IP address from a shared IP address pool. Although the IP allocation should be random, a malicious customer can obtain the desired IP address by repeatedly allocating and releasing IP addresses.

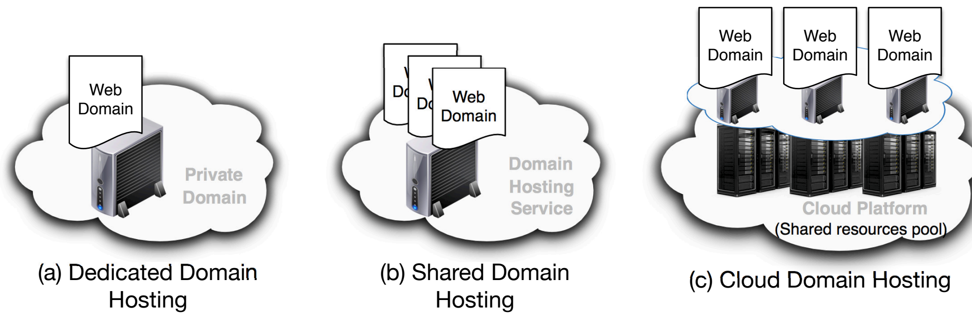


Figure 5.5: Three paradigms of modern domain hosting.

Therefore, we focus on the security threat of Dare-A in the context of cloud environments, especially the two most popular cloud platforms, Amazon EC2 [2] and Microsoft Azure [30].

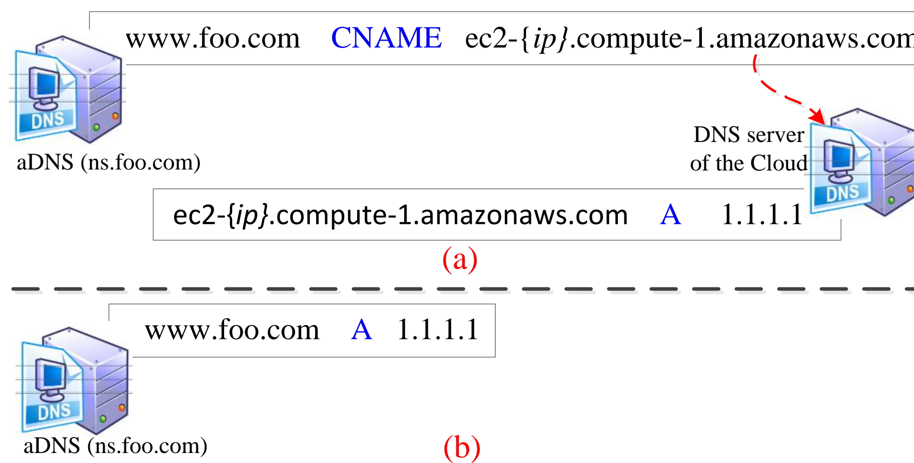


Figure 5.6: aDNS setups for a domain hosted in the cloud.

Amazon EC2. In Amazon EC2, users can rent virtual machines (instances) and run their own applications. By default, when an instance is initiated, it will be assigned a public IP address; when the instance is terminated, it will release the assigned IP address. EC2 also provides Elastic IP, a persistent public IP address allocated to a user's account. An Elastic IP is held by a user until she releases it. Once

an Elastic IP is released, it will be recycled by EC2 and becomes re-allocable to other users immediately. Each instance that receives a public IP address is also given an external hostname in the form of `ec2-{ip}.compute-1.amazonaws.com`. Moreover, EC2 currently provides two types of platforms: EC2-Classic and EC2-VPC [3]. While they differ in many aspects, the major difference that matters to us is that a separate public IP address pool is used for each type of platform. According to Amazon [44], all EC2 accounts created after December 4, 2013 can only use EC2-VPC, while EC2-Classic is merely available for accounts that have used it before on a region-by-region basis.

Once a public IP address in the cloud is obtained, a user can point its domain resource (e.g., a web server) to the IP address using either `CNAME` or `A` record, as illustrated in Figure 5.6. Once adversaries successfully obtain the IP address of a Dare-A, they are able to impersonate the domain resource at their will, regardless of which EC2 platform the domain resource resides in and which kind of DNS record it uses for pointing.

Microsoft Azure. Similar to EC2, the public IP addresses on Azure also fall into two categories: dynamic and reserved. A dynamic IP is allocated and then released when its associated resource, such as a virtual machine, is initiated and then terminated, respectively. To prevent its IP address from changing, a user can explicitly reserve an IP address, i.e., a static IP address. Our measurement shows that both types of public IP addresses are allocated from the same IP address pool, and any one of them becomes re-allocable immediately after being released. Furthermore, a dynamic IP can be converted to a reserved IP under a user's demand. Finally, to point a domain resource to a public IP address on Azure, the same simple technique (i.e., using either `CNAME` or `A` record) is applied.

Dares	IP in Cloud	Abandoned Services	Expired Domains
Dare-A	√	√	
Dare-CN	√	√	√
Dare-MX	√	√	√
Dare-NS	√		√

Table 5.2: Summary of the attack vectors to which each type of Dare is vulnerable.

5.3.3 Abandoned Third-party Services

Modern websites extensively use third-party services. For instance, they may use Mailgun [29] for email delivery and Shopify [38] for building online retail point-of-sale systems and stores. These services usually provide users a subdomain where the corresponding service is hosted. For example, when a user, Alice, subscribes the service from Shopify, she will be assigned a subdomain name, `alice.myshopify.com`, and thus her online store is accessible via this subdomain. However, in most cases, people prefer to have their stores under their own domains. To this end, each third-party service allows users to point their (sub)domains to the resource provided by the service using `A` or `CNAME` records. In the example of Shopify, Alice can set up her aDNS as follows:

```
     shop.Alice.com A 23.227.38.32
(or) shop.Alice.com CNAME alice.myshopify.com
```

In addition, Shopify's DNS server resolves all users' subdomains to a dedicated domain:

```
*.myshopify.com CNAME shops.shopify.com
```

Since all custom domains of Shopify point to the same IP address (`23.227.38.32`) or the same domain (`shops.shopify.com`), Alice also needs to claim ownership of `shop.Alice.com` on her Shopify account. In this way, Alice's store can be accessed through `shop.Alice.com`.

Later, when Alice does not want to use Shopify anymore, she can stop the service and purge the above DNS records. However, if she forgets to do the cleaning, `shop.Alice.com` will continue to be resolved to `shops.shopify.com` since most services use a wildcard to resolve user-specific subdomains (as Shopify does). Now, if an

adversary, Malice, knows that `shop.Alice.com` points to Shopify, he can claim ownership of it. If Shopify does not verify the claim, which is a common practice in most services, Malice can now control the subdomain, `shop.Alice.com`.

The case of email service is similar to this process. The only difference is that a user adds `MX` records, instead of `CNAME` records, for receiving emails.

A verification of domain ownership can prevent the above attacks. However, in some cases, verification is too costly, if not infeasible. For example, the Azure cloud service employs a user-specified subdomain naming scheme. To achieve the domain ownership verification, the cloud has to remember all subdomain names previously used by each user, and the induced cost is prohibitively high considering the large scale of Azure's user group.

In summary, for this attack to succeed, it requires that:

- the vulnerable domain can be resolved to a common target (e.g., IP address or domain name) and the third-party service does not verify the ownership of the vulnerable domain; *or*
- the vulnerable domain resolves to a custom target that can be obtained by any user when it is available.

5.3.4 Expired Domains

The `data` fields of `CNAME`, `MX`, and `NS` records may point to expired domains. An adversary may re-register and abuse the expired domains. Our attack differs from previous works [141], [161], [170] in that they mainly exploit the residual trust of the expired domains, while ours abuses the trust of the *unexpired* (sub)domains pointing to the expired domains. Such stale records are pervasively neglected by domain administrators because (1) there could be secondary records as a means of failover (e.g., multiple `MX` and `NS` records); and (2) the services linking to the expired domains are no longer used and no one cares about updating them.

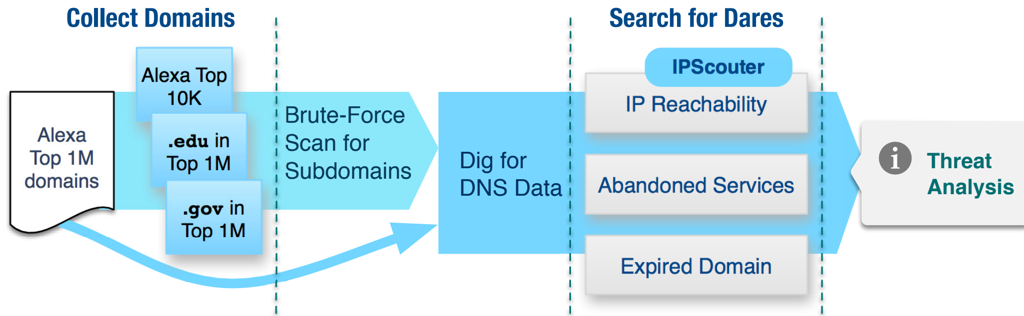


Figure 5.7: Methodology overview.

5.3.5 Summary

Overall, we have shown that several types of Dares can be exploited in multiple ways. Table 5.2 summarizes the attack vectors to which each type of Dare is vulnerable.

5.4 Measurement Methodology

To assess the magnitude of the Dares problem, we conduct a large-scale measurement study. The overview of our measurement methodology is shown in Figure 5.7. We attempt to answer the following two questions: (1) How prevalent is each type of Dares in the wild? and (2) what are the security implications of Dares?

5.4.1 Domain Collection

In order to comprehensively detect Dares, it is ideal to collect the DNS data for all apex domains and their subdomains. However, it is impractical to scan all domains. Since only popular ones could pose a serious threat, we build our dataset as listed in Table 5.3. We first obtain a list of apex domains. Here we choose a snapshot of Alexa's top 1 million domain list for each year from 2010 to 2016. These top domains are particularly attractive because a popular domain provides higher value if an adversary can control it. This set of domains is denoted as D. Note that our dataset D in essence is different from the expired domains studied in [141]. In the case of expired domains, the DNS records in all resolvers were purged, resulting in no Dares.

It is also non-trivial to obtain the complete subdomains of an apex domain. Since the majority of apex domains disallow a DNS zone transfer (i.e., a DNS query

Dataset	Data Space
D	Unexpired apex domains in Alexa top 1M during 2010 $\sim$ 2016
S_t	Subdomains of top 10,000 general
S_e	Subdomains of top 2,700 .edu
S_g	Subdomains of top 1,700 .gov

Table 5.3: Evaluation set of domains.

of type AXFR), we decide to use brute-force scanning to construct our subdomain list. However, it is impractical to scan all top 1 million domains. To make this search manageable, we constrain our search space to the first 10,000 domains, 2,700 `.edu` domains, and 1,700 `.gov` domains in the top 1 million domain list. We first issue a DNS zone transfer query to each of these domains, and we successfully collect the zone data for 320 domains. Based on the results of zone transfers, we then construct a word list of size 20,000 for brute-force scanning. The zone transfer results also show that the wildcard records (e.g., `*.foo.com`) are widely used in practice. In our brute-force scanning, we carefully eliminate the non-existent subdomains. In this process, we send DNS queries to about 288 million valid subdomains and about 570 thousand subdomains are successfully obtained. This subdomain dataset is denoted as $S = S_t \cup S_e \cup S_g$.

5.4.2 DNS Data Retrieval

Then we use the DNS tool `dig` to retrieve the DNS records of every domain in D and S. We only collect the DNS records whose types are listed in Table 5.1. For these types of DNS records except `A` record, we recursively issue DNS queries for the hostname in the `data` field until a query reaches (or fails to reach) an `A` record. Therefore, for each domain d in $D \cup S$, we obtain a DNS resolving chain $RC_d = \{rtype_0(d, data_0),\ \ldots,\ rtype_i(data_{i-1}, data_i)\}$. This dataset is denoted as DREC $= \bigcup RC_d$.

5.4.3 Searching for Dares

After the completion of DNS data collection, we then automatically search for the four types of Dares using Algorithm 1. Given the resolving chain of a domain, we

recursively check the `data` field of every DNS record in the chain, as shown in Lines 7, 9, 12, 15 and 19 of Algorithm 1. The Dare type is determined by the type of the first DNS record of the chain. We next describe how we implement these checks in detail.

Algorithm 1 Search for Dares.

```
Input: DREC, ALLOCIP
Output: Dares (DARES) and potential Dares (PDARES)
 1: procedure DAREFINDER(DREC, ALLOCIP)
 2:     for RC ∈ DREC do
 3:         daretype ← RC.rtype_0
 4:         for rec ∈ RC do
 5:             hostname, rtype, data ← unpack(rec)
 6:             if rtype == "A" then
 7:                 if data ∈ ALLOCIP then
 8:                     DARES ← [daretype, rec, data]
 9:                 else if likely_dareA(data) then
10:                     PDARES ← [daretype, rec, data]
11:             if rtype ∈ ["CN", "MX"] then
12:                 if domain_expired(data) then
13:                     DARES ← [daretype, rec, data]
14:                     break
15:                 if abandoned_service(data) then
16:                     DARES ← [daretype, rec, data]
17:                     break
18:             if rtype == "NS" then
19:                 if domain_expired(data) then
20:                     DARES ← [daretype, rec, data]
21:                     break
```

5.4.3.1 Checking A Records (Lines 7 and 9)

Both EC2 and Azure publish their public IP ranges [6], [31]. However, we still cannot know if a given IP is allocable at a specific time. Almost all cloud platforms including EC2 and Azure assign IP addresses randomly and disallow users to specify an IP to allocate. It is a challenging task to obtain a desired IP. We study this issue from the following two aspects:

- We quantify whether and how practical an attacker can overcome the random IP assignment to obtain a desired IP by scouting the IP pools.
- We then assess the potential magnitude of Dares in the wild.

Scouting IP Pools. We implement a simple tool, IPScouter, to milk the IP addresses from EC2 and Azure. Since EC2 uses two separate address pools for EC2-Classic and EC2-VPC, we set up two IPScouters, one for each address pool. The

IPScouter-VPC randomly requests the IPs from all currently available regions [34]. The IPScouter-Classic requests the IPs from `us-east-1` as our account can support EC2-Classic only in this region. In both setups, only Elastic IPs are allocated via the `boto`'s [8] API `allocate_address`. Also, the IPScouter-Azure requests the IPs from the regions returned by `ServiceManagementService.list_locations()` [7]. The static IP addresses are reserved by using `create_reserved_ip_address()`. No virtual machine or service is launched in this process.

The obtained IP addresses are immediately released after being logged into `ALLOCIP`, the input to Algorithm 1. Finally, since all clouds throttle query API requests on a per-account basis, our IPScouters employ the exponential-backoff-linear-recovery strategy to control their request rates.

Potential Dares in the Wild. Our IPScouter is probabilistic by nature and many factors could affect the completeness of the milked IPs. We may not find all desired IPs in our study. For example, a cloud platform may reserve a portion of IP ranges for some time. Therefore, we scan all IPs in `DREC` to assess the potential number of exploitable Dares in the wild. Our basic assumption is that if an IP in a cloud is not alive, it has probably been released. In both EC2 and Azure, an in-use IP costs nothing, but users should pay for an unused one. Thus, we believe this assumption is valid in general. Given a set of `A` records $R = \{r_1, r_2, \cdots, r_n\}$, with r_i = <$\texttt{name}_i$, $\texttt{IP}_i$> and $i \in [1, n]$, we check if they are potential Dares based on the following steps (Line 9 in Algorithm 1):

Step 1. If IP_i is not in the cloud, remove r_i.

Step 2. We remove all records that are very unlikely to be dangling based on their `name` fields. For instance, a record may point to a specific service built atop an existing IaaS infrastructure like load balancing. These records are usually managed by the cloud DNS servers. If the DNS resolution is successful, it indicates that the IP is not released.

Step 3. We scan the remaining records using ZMap [104]. To reduce the

scanning traffic, we prioritize the ports using a set of heuristics. For instance, the ports for HTTP and HTTPS are ranked first by default. If the `name` field starts with `ns`, it is probably a DNS server, and thus we scan port 53 first using both TCP and UDP. Note that we conduct a second scanning for all non-alive IP addresses after one month to ensure they are not transient failures.

Step 4. At this step, all remaining records are probably Dares since they are associated with these not-in-use IP addresses. We further check `archive.org` to gain more confidence on whether an archived webpage for $\texttt{name}_i$ can be found (see §5.5.2).

5.4.3.2 Checking Abandoned Services (Line 15)

We first identify a list of popular third-party services. To this end, we cluster all `CNAME` and `MX` records based on their `data` fields and then manually check all email and top 200 non-email services in terms of the cluster size. A service is selected for further checking if it (1) satisfies one of the two requirements in §5.3.3 (i.e., the domains using the service are vulnerable to security-sensitive Dares) and (2) provides free or free-trial accounts, which allow us for further checking. As listed in Table 5.4, only one email and eight non-email services meet the two pre-conditions and are chosen for further checking. The majority of those non-selected services do not provide free accounts to individuals, preventing them from being further checked. For non-email services, only several, such as Google [19] and Aliyun [1], enforce ownership verification. By contrast, we find only one email service that does not enforce ownership verification. This is probably because most email service providers try to prevent their services from being abused in spamming and phishing. A common practice of verification requires domain owners to include a random `CNAME` or `TXT` record into their aDNS's records. Since we assume that adversaries cannot control a domain's aDNS, such verification will be able to foil all attack attempts.

Then, to automatically find unclaimed domains, we build an automated tool by leveraging Selenium [37], a tool that automates web browsing. Note that we can easily single out unclaimed domains in Azure by simply finding these `CNAME` records whose

Type	Service List
`CN`	Azure cloud service (`cloudapp.net`), Shopify, Github, Wordpress, Heroku, Tumblr, Statuspage, Unbounce
`MX`	Mailgun

Table 5.4: Evaluated third-party services.

`data` fields fail to be resolved to `A` records. There is no need to use the automated tool because the domains in the `name` fields of these `CNAME` records should be unclaimed in the Azure cloud service.

5.4.3.3 Checking Expired Domains (Lines 12 and 19)

It is straightforward to check whether a domain has expired. We first screen out the expired domains based on the `WHOIS` responses. For an expired domain, the response from `WHOIS` should be null. Since `WHOIS` is not always reliable, we then crosscheck with the popular Internet domain registrars such as GoDaddy to verify the expiration of a domain if we can re-register the domain.

5.4.4 Limitations

While our work is able to find exploitable Dares in the wild, we cannot know whether and how many websites have already been exploited. For example, an expired domain may have already been registered by an attacker. Moreover, our study currently covers only two cloud platforms and nine third-party services. However, the Dare problem should be universal across many cloud platforms and third-party services.

5.5 Measurement Results

In this section, we demonstrate that the problem of Dares is widespread and underplayed, even in those well managed zones like `edu` and `gov`. We first describe the general characteristics of Dares found in our measurement study and then analyze the measurement results with respect to the three attack vectors.

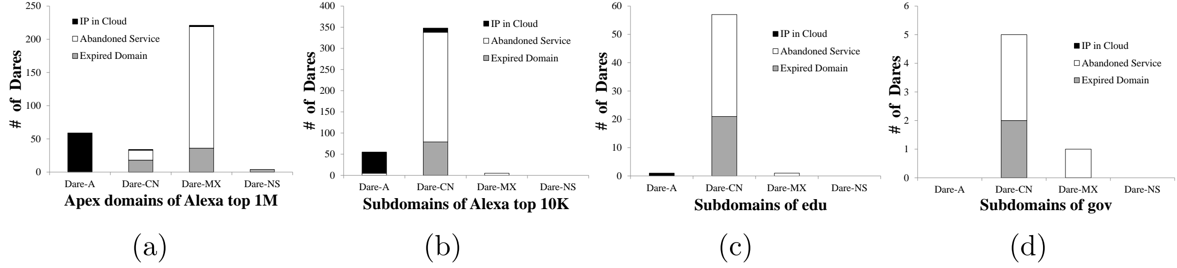


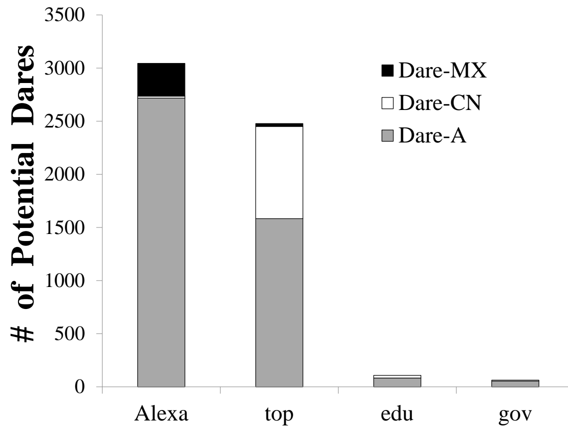
Figure 5.8: Number of confirmed Dares for each dataset.

5.5.1 Characterization of Dares

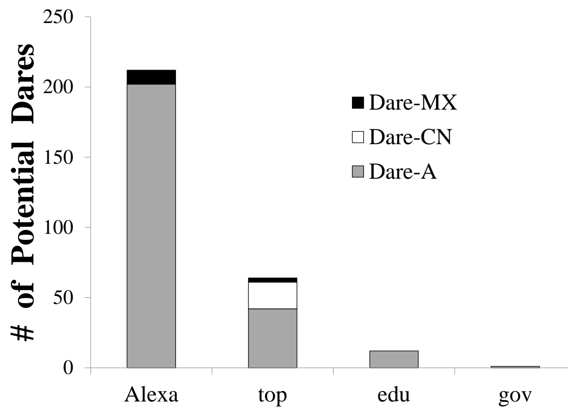
Figure 5.8 presents the number of Dares found in the four datasets listed in Table 5.3. In this figure, we only count the IP addresses that are successfully obtained by our IPScouters (i.e., confirmed Dares). The remaining potential Dares are presented in Figure 5.9. For dataset `S`, multiple Dares in the same domain zone are counted separately. In total, we find 791 Dares and 5,982 potential Dares in the wild. As we can see, Dares exist in all four datasets, indicating a widespread problem.

It is evident that the total number of Dare-A and Dare-CN accounts for the majority of confirmed and potential Dares in the wild. This is because `A` and `CNAME` records are the most frequently used in practice. For apex domains, more than 90% delegate their aDNSes to third-party services like GoDaddy [112]. When the hosting resources are released after a website is closed, its aDNS is usually still alive and all DNS records will unlikely be deleted as the domain itself is still unexpired. The `A` and `CNAME` records for subdomains commonly link to new resources supported by the domains or external services, which often have a relatively short lifetime and can sometimes be migrated away. Due to the high churn rate of these subdomains, it poses a tedious burden for domain owners to manually keep their aDNS servers updated and consistent. Therefore, in practice, these stale DNS records are usually not purged, resulting in Dares. Note that the number of Dare-CNs in dataset D is relatively small because it is normally not recommended to keep `CNAME` records at apex domains.

Dare-MX is mainly caused by abandoned services, and only dataset D has instances that can be exploited by the other two vectors. After examining these special



(a) Amazon EC2



(b) Microsoft Azure

Figure 5.9: Number of potential Dares.

instances, we find that domains in D tend to use multiple `MX` records pointing to different domains. For example, the DNS records of domain `customizedgirl.com` include

```
customizedgirl.com@ns-1057.awsdns-04.org.:
    customizedgirl.com. 60 MX 10 bridalpartytees.com.
    customizedgirl.com. 60 MX 10 customizedgirl.com.
    customizedgirl.com. 60 MX 10 shoplattitude.com.
```

Here three `MX` records point to three different domains with the same priority. We find that the third one, `shoplattitude.com`, has expired. We speculate that this is a typo, which should actually be `shoplatitude.com`. Since each of the three records is used in a round-robin fashion by resolvers, it is difficult for domain owners to quickly be aware of the failed record. By contrast, all `MX` records in a better-managed domain

like those in dataset S usually point to different mail servers of the same domain.

Finally, only four instances of Dare-NS are found in our measurement, all of which are in dataset D. All instances share the same pattern of misconfiguration, with one example shown below.

```
bedshed.com.au@ns1.partnerconsole.net:
    bedshed.com.au. 3600 NS ns2.r2design.com.au.
    bedshed.com.au. 3600 NS ns1.r2design.com.au.
```

Using `dig` utility with `+trace` option, we find that the actual aDNS in `.com.au` TLD is `ns1.partnerconsole.net`, but the `NS` records are not updated and still point to the expired domains. The smidgen of Dare-NS in the wild is probably because the `NS` records are more critical, and a misconfiguration can be easily spotted. Moreover, the majority of domains have migrated aDNS to third-party services [112], which usually have well-managed servers. Unfortunately, this migration also becomes a common cause of Dare-A that can be exploited through the IP in cloud (see §5.5.2).

Dare	**top 10K** (S_t)	**edu** (S_e)	**gov** (S_g)	
Dare-A	40	1	0	
Dare-CN	260	50	5	
Dare-MX	5	1	1	
Total	277†	52	6	335

Table 5.5: Statistics of distinct apex domains in `S` with confirmed Dares. † Some domains overlap across the above three lines (e.g., a domain has both Dare-A and Dare-CN).

For dataset `S`, Table 5.5 shows the number of distinct apex domains for each type of Dare. In total, we identify Dares for 277 distinct domains in Alexa's top 10,000, 52 in `edu` zone and 6 in `gov` zone. In particular, the domains in S_t cover many types of websites as shown in Figure 5.10. Our results demonstrate that Dares indeed exist in almost all types of websites and thus can incur serious damages.

5.5.2 IP in Cloud

We now analyze the measurement results of the first attack vector in two cloud platforms: Amazon EC-2 and Microsoft Azure.

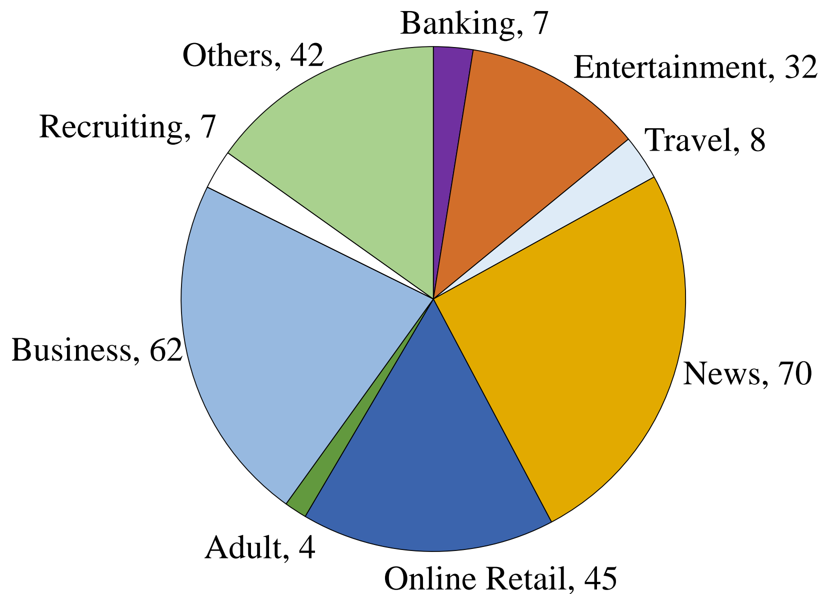


Figure 5.10: Categories of websites that have Dares.

Performance of IPScouters. Figure 5.11 shows the number of distinct IP addresses obtained by IPScouters over time. IPScouter-Classic and IPScouter-Azure last for 14 days and IPScouter-VPC lasts for 26 days. For EC2-VPC and Azure, the number increases linearly, with about 5,000 and 2,200 new IP addresses obtained daily, respectively. However, the number on EC2-Classic only rapidly increases in the first few days and then stops growing over time. The speed of IPScouters is mainly constrained by three factors: the request rate limit of the clouds, the randomness of IP allocation, and the density of IP address space. For the first constraint, we find that a five-second delay between two API calls (IP allocation or release) works fine with EC2, but at least a ten-second delay should be used in Azure. Under this configuration, IPScouters send about 7,900 and 4,300 IP allocation requests per day to EC2 and Azure, respectively.

Although it seems that IP allocation is not truly random, it is very unlikely for a cloud to re-use the recently released IP addresses. That is why on both EC2-VPC and Azure the number of daily obtained new IP addresses remains about half of the number of allocation requests sent to the clouds. This speed is fast enough to feasibly milk a large number of IP addresses in each cloud.

The significant speed decrease on EC2-Classic is probably due to the crowded

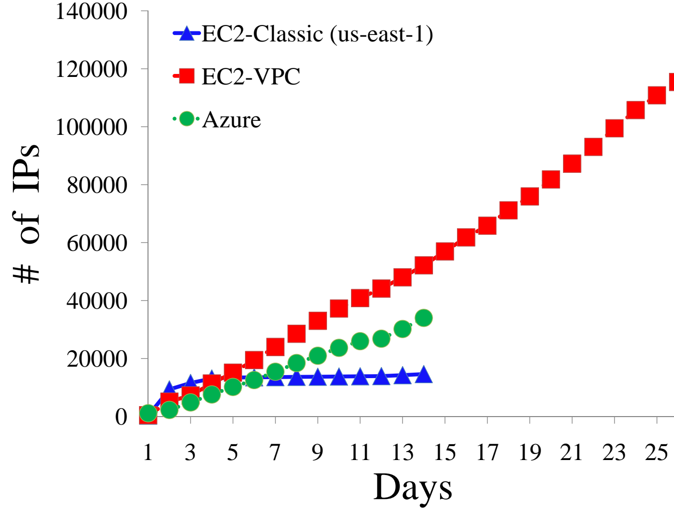


Figure 5.11: Number of IP addresses milked on clouds over time.

IP address space. For instance, for all those domains that use EC2 in our datasets, about 69% are hosted on EC2-Classic. By contrast, both EC2-VPC and Azure have larger address space but fewer users. Note that, although the IP address spaces of EC2 and Azure include millions of IP addresses [6], [31], we speculate that only a portion of IP address space is available at any time.

We deploy only one IPScouter for each cloud platform, while adversaries may deploy IPScouter farms to significantly speed up the IP milking. Finally, the IP allocation in clouds is a complicated issue that deserves in-deep study, and we leave this exploration as our future work. In this work, our goal will be to demonstrate that the Dare problem is a real and serious threat.

Confirmed and Potential Dares. In our measurement, all confirmed Dares are from EC2, with about 93% from EC2-Classic. Considering that IPScouter-Classic milks IP addresses from only one region, more potential Dares would have been confirmed if we extended our search to other EC2 regions. Meanwhile, as shown in Figure 5.9, the number of potential Dares on EC2 is significantly larger than that on Azure. This is because Azure is a relatively new platform and has a much smaller market share than EC2. For instance, we find that among all domains that use clouds in our

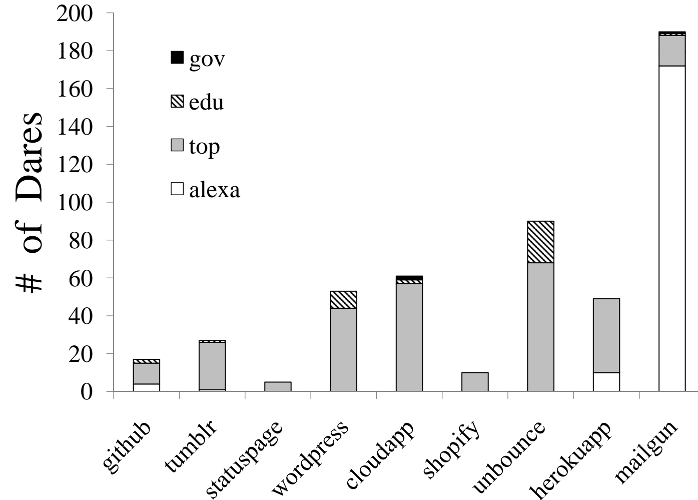


Figure 5.12: Number of Dares on each third-party service.

datasets, Azure hosts just one tenth. Thus, more time is required to milk desired IPs of Azure. However, this does not reduce the generality of a potential attack; the problem is universal. As the clouds become more crowded, the threat will be more serious and widespread.

By further crosschecking with `archive.org`, we successfully find snapshots for about 52.6% potential Dares. Thus, these domains can be claimed as true Dares with higher confidence.

We only identify a few potential Dares in `edu` and `gov` zones. Domains in these zones are mostly deployed using the paradigm of Figure 5.5(a), where no cloud IP is used. Besides, the majority of domains in `gov` zones use Rackspace [33], instead of EC2/Azure.

Patterns of Dares. As shown in Figures 5.8(a) and 5.9, this attack vector can effectively exploit both apex domains and subdomains. We attempt to infer how the Dares are introduced by manually searching and checking relevant information of all confirmed Dares and 100 randomly sampled potential Dares.

While many vulnerable apex domains are toy websites with low value, more than half of those identified belong to startups for which we can find company information

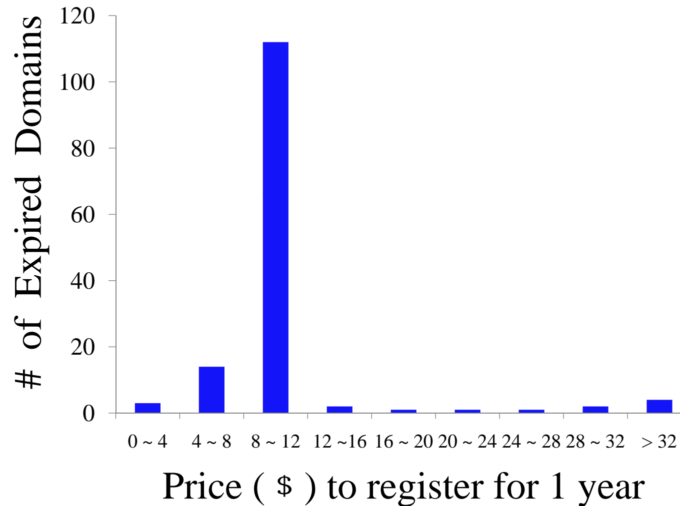


Figure 5.13: Prices to register the expired domains for one year.

on CrunchBase [9], Twitter, and Github. One of the examples is described in §5.6.1. These startups are either closed, re-branded, or acquired by other companies. In all cases we examined, although the domain owners have released the hosting resources in a cloud, they continue to renew their domains. Such vulnerable apex domains provide valuable and attractive properties for attackers to conduct phishing and scamming.

For those vulnerable subdomains, we uncover two main causes of Dares. First, Dares are introduced due to website re-construction. One such example is `support.mediafire.com`. Previously, the "Get Support" on its homepage was linked to `www.support.mediafire.com/help`. However, it now points to the link of `www.mediafire.com/help`, and the host in the cloud for `support.mediafire.com` has been released. Clearly, the domain owner forgot to update their DNS servers with this change. In another example, `autotrader.co.uk` stopped the self-managed aDNS (`ns4.autotrader.co.uk`) and delegated the aDNS resolution to `verisigndns`. After this delegation, although they correctly updated the `NS` records, they forgot to delete the glue records. The second cause is simply that certain services have been discontinued. For instance, `books.panerabread.com` previously collaborated with Amazon to sell books. This service now seems to be closed. Again, the hosting resources are released, but its

aDNS is not updated.

Cost analysis. During our study, IPScouters cost about \$0.07 on EC2 and \$0.005 on Azure per day. Such a low cost makes it feasible to conduct long-term IP milking. Once a desired IP is obtained, to hold it with a cheapest virtual machine, it costs \$0.0115 and \$0.023 per hour or \$100.74 and \$201.48 annually on EC2 and Azure, respectively. By contrast, the same expense would afford adversaries only several minimally effective typosquatting domains.

5.5.3 Abandoned Third-party Services

Figure 5.12 presents the number of Dares found on each third-party service, showing that Dares can be found on every service platform. Most Dares on Mailgun are in dataset D because email services are commonly hosted under apex domains. Instead, non-email services usually serve as the sub-functions of an apex domain and thus reside in the subdomains. We find that this Dare problem is quite worrisome as Dares can even be found in famous domains like `Yahoo.net` and `mit.edu`.

Patterns of Dares. While most Dares occur because the third-party services are abandoned, we find an interesting pattern in one of the services, Wordpress, as shown in the following example.

```
www2.opensky.com@ns-1448.awsdns-53.org.:
www2.opensky.com. CNAME blog.opensky.com.
blog.opensky.com. CNAME openskymerchants.wordpress.com.
```

The website of `blog.opensky.com` is still in use and its original webpage can be reached. The domain owner intends to direct `www2.opensky.com` to `blog.opensky.com` using `CNAME`. Unfortunately, this configuration fails to function properly and accessing `www2.opensky.com` will reach an error page on Wordpress. The problem lies in the fact that only `blog.opensky.com` is claimed on Wordpress, which dispatches web requests according to the initial domain name. Since `www2.opensky.com` is not claimed, Wordpress will direct all requests to the error page. An attacker can thus claim the subdomain, and all subsequent requests will then be redirected to the landing page under the

attacker's control, although the CNAME tries to redirect to blog.opensky.com. While we only observe such cases on Wordpress, services like Github, Cloudapp, Shopify, and Herokuapp may also be vulnerable to this misconfiguration. The other three services, including Tumblr, Statuspage, and Unbounce, do not suffer this problem because a subdomain can be claimed if and only if it points to a specific domain like domains.tumblr.com.

Cost analysis. All of these services provide free or free-trial accounts. Thus, it costs adversaries virtually nothing to register many free accounts.

5.5.4 Expired Domains

Patterns of Dares. As our results show, many subdomains in even well-managed zones like edu and Alexa's top domains point to expired domains using CNAME. A further examination reveals three patterns into which these expired domains fall, as listed in Table 5.6. First, more than one-third of expired domains look quite similar to their alias subdomains. For instance, module.rabobank.nl points to rabobank-hoi.nl and rps.berkeley.edu points to rpsberkeley.org. Second, as found in [141, 170], a significant portion of subdomains point to expired external services. One example is 21vcdn.com. The subdomain, js.jiayuan.com, points to the service that has stopped working since 2010. Third, we find several cases of typos. For instance, b.ns.trnty.edu points to awsnds-18.net. The domain owner obviously intends to use a CNAME record to redirect their previous aDNS to the one provided by Amazon AWS. This attempt fails because of a typo. The domain currently uses NS records to point to Amazon AWS directly, but the mistyped CNAME record still exists. The remaining 33% of expired domains basically comprise random characters.

Existing defense against abusive domain registration. We have re-registered all the expired examples listed in the dissertation (i.e., eight expired domains). After about three months, our re-registered domains are still alive and we received warning from only one domain owner. This indicates that the majority of expired domains are indeed vulnerable to be abused. Domain registrars and owners

Pattern	Examples	%
Similar to alias	module.rabobank.nl → rabobank-hoi.nl rps.berkeley.edu → rpsberkeley.org	39%
Expired external services	js.jiayuan.com → 21vcdn.com shopping.segye.com → ticketdamoa.com	21%
Typo	b.ns.trnty.edu → awsnds-18.net customizedgirl.com → shoplattitude.com	7%

Table 5.6: Patterns of expired domains.

may adopt existing defense mechanisms to protect against the registration of abusive domains. First, they can disapprove those domains in malicious domain lists. However, we find that none of our identified expired domains are included in these lists. Second, they can disallow domain names that are very similar to well-known ones to be arbitrarily registered. In our datasets, we identify that this can prevent about 46% of expired domains from being exploited. Unfortunately, still about 54% of expired domains are irrelevant to vulnerable subdomains. It is difficult for registrars to determine whether such an expired domain is associated with Dares, rendering these misconducts hard to be thwarted. Therefore, more effective defense is needed to prevent abusive domain registration.

Cost analysis. These expired domains are also quite cheap to own. Figure 5.13 shows the prices to re-register these domains for one year. It costs less than $12 for most domains. Given the significant value of these vulnerable subdomains, this cost is negligible.

5.5.5 Exploiting Dares

We now determine the exploitable window of Dares. For those caused by released IP addresses in clouds and abandoned third-party services, we estimate their occurrence time by checking with `archive.org`. For expired domains, we can find their expiration date. Our results show that all Dares have a large exploitable window, ranging from three months to seven years, with over 90% being vulnerable for more than one year.

5.5.6 Ethical Considerations

During the process of this study, we did not conduct any adversarial activities against the scrutinized domains or the visitors to the Dares we successfully identified. We also checked with our institution's IRB and confirmed that we do not need to obtain its approval. All examples presented in this dissertation have either been corrected or defensively exploited by us. We have sent notifications to all affected domains, and have received responses from roughly half of them. Almost all apex domains did not reply. Although most subdomains have acknowledged our reports, only two thirds of them have taken action for remedy. Our experience is similar to the observations by Li et. al. [143].

5.6 Threat Analysis

Domain names serve as the trusted base in many security paradigms. For example, human users and many malicious domain detectors tend to assume an apex domain with a clean history as trusted. A user also trusts all subdomains of an apex domain with good reputation by nature. Unfortunately, our work demonstrates that such trust could be abused by adversaries to mount a number of much more powerful attacks. In this section, we describe and discuss four types of threats that could be significantly exacerbated by exploiting Dares.

5.6.1 Scamming, Phishing, and More

The common modus operandi that adversaries adopt in scamming, phishing, and many other forms of malicious activities includes typosquatting [131], doppelganger domains [13], and homograph attacks [118]. However, these approaches are limited in effectiveness, and vigilant users can easily spot them. Moreover, many automatic systems like EXPOSURE [74] and Notos [65] have been proposed to detect these malicious domains.

Dares can significantly enhance the effectiveness of these malicious attacks in two major ways. First, instead of registering new domains, adversaries directly abuse either

subdomains or apex domains usually with a clean history and an excellent reputation. The abused domains have unchanged registration information and can even reside on the same IP addresses. Second, at an affordable cost, adversaries can target a large number of victims in a short time by leveraging services like Google AdWords. We next illustrate three case studies.

Case 1: Suspended domains getting revived. `GeoIQ.com` [17] is a web-based location analysis platform offering data sharing, risk mitigation, and real-time analysis services. The A record retrieved from its aDNS is shown below. We can see that this domain was hosted on EC2.

```
geoiq.com@ns-1496.awsdns-59.org.:
    geoiq.com. 1800    A    23.21.108.12
```

In July 2012, `GeoIQ.com` was acquired by another company and `archive.org` shows that the last snapshot of this domain was captured on August 1, 2015. This implies that the domain owners released the hosting resources in EC2 around August 2015, which was later successfully obtained by our IPScouter. However, the `WHOIS` data shows that the domain still gets renewed annually.

```
Domain Name: GEOIQ.COM
Registrar: GODADDY.COM, LLC
Updated Date: 21-sep-2015
Creation Date: 20-sep-2005
Expiration Date: 20-sep-2016
```

With a simple Google search, we can find their accounts on many platforms, including Github, Twitter, and Youtube. Adversaries could impersonate the domain and launch social engineering attacks more effectively.

Case 2: Inherited trust from apex domains. `mediafire.com`, ranked 169 in Alexa at the time of our study, is a file hosting, file synchronization and cloud storage service provider. One of their subdomains, `support.mediafire.com`, was hosted on EC2 but later was no longer used. The hosting service on EC2 was released and then successfully obtained by our IPScouters.

```
support.mediafire.com@ns-1179.awsdns-19.org.:
```

```
support.mediafire.com. 86400 A 23.21.94.181
```

The subdomain `support` is a common practice used by many domains to provide supporting services to users. There are also many other similar cases like `jobs`, `payment`, or `shop`. If an adversary hosted malicious contents or carried out spear phishing under such subdomains, even the most vigilant users would fall victim to the attacks.

Case 3: Harvesting through Google Adwords. `Travelocity.com`, ranked 1,810 in Alexa at the time of our study, is one of the largest online travel agencies. We find that one of its subdomains points to an expired domain using `CNAME` record.

```
can.travelocity.com@pdns1.ultradns.net.:
  can.travelocity.com. CNAME travelocitycancontest.com.
```

To demonstrate how fast an adversary can spread the attacks and at what cost, we register this expired domain and direct visitors to our subdomain using Google AdWords. To minimize the inconvenience that our study might have caused, we redirect all visitors to the homepage of Travelocity after recording the MD5 of source IP addresses. Since our interaction with the users is limited to logging the hashed IP addresses, we believe there are no ethical implications in this experiment. We run the campaign for two days, and 141 distinct IP addresses are recorded at the cost of $1.38. Adversaries could set up a fake login page or steal cookies directly. In either case, thousands of accounts could be compromised.

5.6.2 Active Cookie Stealing

Adversaries have multiple ways to steal and hijack cookies. One simple approach requires the traffic between users and websites to be unencrypted and adversaries to be able to monitor the traffic. This strong requirement limits the scale and feasibility of this approach for cookie stealing. For instance, almost all top websites have adopted at least partial HTTPS [191] and sensitive cookies are usually transmitted in HTTPS only (using the `Secure` flag). Alternatively, if HTTP cookies do not have the `HttpOnly`

flag set, adversaries can obtain them through other means like XSS attacks. As this flag is being deployed on more websites, XSS attacks will become ineffective in cookie hijacking. By exploiting Dares, however, adversaries can actively steal cookies from world-wide users, regardless of the `HttpOnly` and `Secure` flags. This likely results in not only privacy leakage but also fully compromised accounts.

Implications. Whenever possible, cookies with sensitive account information should be scoped to trusted subdomains only. It is also unsafe to rely on the `Secure` flag to prevent cookie stealing. The `Secure` flag is known to lack integrity [229], but it was generally assumed to be secure against stealing. However, this assumption will be challenged by Dares.

5.6.3 Email Fraud

Email is still one of the favorite attack vectors in online fraud. The malicious emails are usually sent with authentic addresses that are not under the adversaries' control. Since adversaries cannot receive and further confirm reply emails from victims, the email attacks are open-looped. However, by exploiting a Dare, an adversary will instead be able to not only send but also *receive* emails. In particular, some popular existing anti-spam mechanisms including Sender Policy Framework (SPF) and DomainKeys Identified Mail (DKIM) can be bypassed. Enhanced with these capabilities, adversaries could conduct many forms of online fraud more effectively and efficiently, from spamming, spear phishing to even abusing exclusive online membership such as Amazon Prime membership.

5.6.4 Forged SSL Certificate

Modern websites commonly provide critical online services over mandatory HTTPS connections, and they allow sensitive cookies to be transmitted only over encrypted connections using the `Secure` flag. For example, the following is a cookie with `Secure` flag set by `travelocity.com`:

```
Set-Cookie: JSESSION=d1b8eb43-xxx; Domain=.travelocity.com; Path=/; Secure; HttpOnly
```

To steal these secure cookies, an adversary has to set up an HTTPS website on the vulnerable subdomain and get it signed by a Certificate Authority (CA). To ensure the authenticity of a certificate, CA usually requires subscribers to prove the ownership of a (sub)domain. This typically involves verification via specific email addresses under the *apex* domain or those in the `WHOIS` database. Adversaries in our threat model can hardly complete this verification.

However, the emerging new Certificate Authority, such as `Let's Encrypt` [24], tends to leverage the *automated* and *free* validation to simplify the process of issuing certificates. `Let's Encrypt` provides two ways for subscribers to prove the control of a domain, one of which involves provisioning an HTTP resource under the domain being signed. Unfortunately, when adversaries exploit a Dare through a cloud IP or an expired domain, they have the full access to the hosting resource of the domain and thus can pass the challenge of `Let's Encrypt`. Using this principle, we successfully have a subdomain `can.travelocity.com` [40] authentically signed.[1]

Implications. It is insufficient to use merely one single challenge for ownership verification. Considering that both aDNS (in the case of Dare-NS) and domain hosting resources could be compromised, it would seem more reliable to seek confirmation from specific emails, e.g., those in the `WHOIS` database.

5.7 Mitigations

Almost all previous efforts, such as the Domain Name System Security Extensions (DNSSEC), attempt to protect the integrity and authenticity of DNS records returned to clients. Little attention has been paid to authenticating the resources to which DNS records point. Domain owners are commonly assumed to keep their aDNS servers updated and consistent. Unfortunately, our work has demonstrated that this assumption rarely holds in practice, and the resulted problem, Dare, is a serious and

[1] The site `can.travelocity.com` is associated with a dangling DNS record and is not currently being used by the domain owner. We temporarily signed the subdomain and directed it to `www.travelocity.com`, and thus there is no break caused by our experiment.

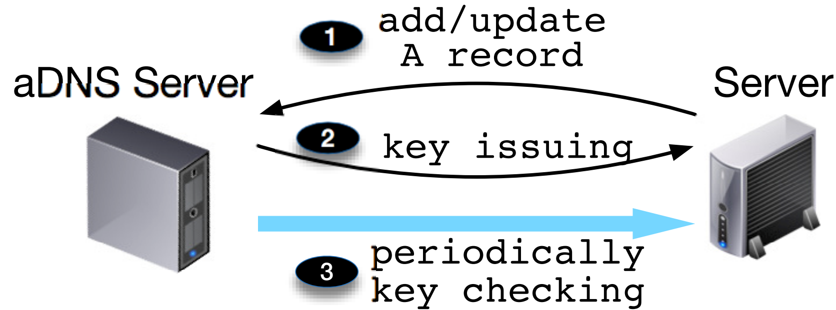


Figure 5.14: Authenticating Ephemeral IPs.

widespread threat. In this section, we propose and discuss the mechanisms that can mitigate Dares with minor manual efforts. In particular, we focus on the DNS `data` fields exploited by our three attack vectors. The key principle of these mechanisms is that *all resources* should be considered *ephemeral*.

Authenticating Ephemeral IP addresses. We propose a mechanism that allows aDNS servers to automatically authenticate IP addresses whenever an `A` record is added or updated. Figure 5.14 shows the workflow of the mechanism. Both aDNS and the corresponding server whose IP address is added/updated have one daemon. When the `A` record is added or updated, the aDNS communicates with the server and issues a key to it. Then, aDNS periodically checks the validation of the key. While the architecture is simple, a set of problems need to be resolved in practice, e.g., how to protect the key on the server and how much overhead is induced on aDNS. We leave the implementation and evaluation of this mechanism as our future work.

Breaking resolution chain through the aDNS of third-party services. In the case of `data` fields pointing to external services, we recommend that services like Shopify should deprecate `A` records and adopt an isolated name space in `CNAME` for each user. In our observations, all the external services except Shopify and Tumblr have deprecated `A` records. To protect the stale `CNAME` records, we define an isolated name space for each user. Since every user already has a unique account number, the services can generate `CNAME` records using the format of {*`user-specified-name`*}.

useraccount.service.com. Multiple domains managed under the same account are assigned unique names like:

```
@aDNS of Shopify
store-1.alice.myshopify.com CNAME shops.shopify.com
store-2.alice.myshopify.com CNAME shops.shopify.com
```

Once the domain of store-1 becomes unclaimed, the record of store-1.alice.myshopify.com should be deleted from the aDNS of Shopify, and thus the dangling domain cannot be resolved.

Checking for expired domains. In existing DNS systems, only the records with expired domains in name fields will be purged from DNS servers, and those with expired domains in data fields (e.g., pointed to by a CNAME) are generally neglected. We have shown that these stale records could be exploited as a major source of Dares. We advocate that aDNS servers should periodically check the expiration of domains in data fields. Since this checking is triggered only when the expiration date is approaching, its frequency is very low and the overall overhead is trivial. Complementary to periodic checking, Alembic [141] can be used to locate potential changes in domain ownership. We are also considering to extend Alembic using the patterns listed in Table 5.6.

5.8 Conclusion

This work studies the problem of dangling DNS records (Dares), which has been largely overlooked, and demonstrates that Dare is a serious and widespread security threat. In order to exploit these unsafe Dares, we have presented three attack vectors, IP in cloud, abandoned third-party services, and expired domains, for domain hijacking. Then we have conducted a large-scale measurement on four datasets containing representative domains to quantify the magnitude of the unsafe Dares in the wild. We have found hundreds of unsafe Dares on even those well-managed zones like edu and Alexa top 10,000 websites. This is very worrisome because Dares can notably enhance many forms of online fraud activities, such as spamming and cookie stealing. The

underlying cause of Dares is the lack of authenticity checking for resources pointed to by DNS records. To this end, we have proposed three defense mechanisms that can effectively mitigate Dares with minor human effort.

Chapter 6

PRACTICAL AND ROBUST DEFENSE AGAINST USE-AFTER-FREE EXPLOITS VIA CONCURRENT POINTER SWEEPING

6.1 Introduction

Memory corruption vulnerabilities have plagued software written in low-level languages like C/C++ for decades. On one hand, effective defenses against previously popular attacks, such as stack and heap overflows [72, 83, 88, 96, 132, 151, 165, 171, 182, 228], have been developed and deployed in commodity systems, thwarting the exploitation of such memory corruption bugs in system software (e.g., browsers or operating systems). On the other hand, recent years have seen the meteoric rise of memory corruption attacks exploiting use-after-free (UaF) vulnerabilities that root in pointers pointing to deallocated memory (i.e., dangling pointers). Actually, UaF vulnerability has become the largest and severest ongoing exploit vector in numerous popular applications [139].

Different approaches have been proposed to harden the memory safety of software against UaF vulnerabilities. Most of the existing solutions attempt to address UaF exploits by making an explicit [163, 166, 168, 188, 221] or implicit [97] safety check on every pointer dereference. An alternative approach is to reshape memory allocators to avoid unsafe memory reuse [59, 72, 171]. Conservative garbage collection [16, 76] heads off UaF exploits through automatic memory management. Moreover, the Silicon Secured Memory (SSM), recently shipped in Sparc M7 processors, implements tagged memory as a hardware UaF defense [12]. Recent works [80, 139, 225] track pointer propagation and nullify dangling pointers at object free.

Unfortunately, these solutions still have two main shortcomings. First, robustness and efficiency cannot be achieved at the same time. UaF exploits are guaranteed

to be defeated but usually with unacceptable or unpredictable overhead [16, 97, 139, 166, 221, 225]. Systems like Cling and SSM incur trivial overhead but provide only partial [59] or probabilistic [12, 72, 171] memory safety. Second, software developers usually cannot obtain sufficient information about the exploited UaF vulnerabilities during production runs, making it difficult to debug and craft patches.

This paper presents a novel defense system, **pSweeper**, which effectively protects against UaF exploits, imposes low overhead for deployment in production environments, and pinpoints the root-causes of UaF vulnerabilities for easier and faster fixing. pSweeper follows a similar protection principle to DANGNULL, DangSan, and FreeSentry [139, 212, 225]. In particular, dangling pointers are proactively neutralized to disrupt potential UaF exploits. However, pSweeper proposes two unique and innovative techniques, concurrent pointer sweeping (CPS) and object origin tracking (OOT) to overcome the above shortcomings.

In order to find and neutralize dangling pointers, all existing works [139, 212, 225] require to *synchronously* keep track of pointer propagation. This design can incur undue overhead, e.g., 80% in DANGNULL [139]. pSweeper instead explores a very different design, **concurrent pointer sweeping (CPS)**, which exploits the increasingly available multi-cores on a computing platform. The core idea is to *iteratively* sweep all live pointers in *concurrent* threads to neutralize dangling ones. The main challenges of implementing CPS are two-fold. First, we must identify and efficiently handle entangled races among pSweeper and application threads. Ideally, we must avoid heavyweight synchronization mechanisms like locks. Meanwhile, we should place heavy workload on pSweeper threads and instrument as few code as possible to application threads. To address this challenge, we leverage hardware features and devise lock-free algorithms that avoid stalling application threads. Second, CPS must scale to massive object (de)allocations and large volume of live pointers. In particular, we must prevent dangling pointers propagating to those swept ones in application threads so that every round of sweeping guarantees to terminate within certain time bounds. To this end, we devise a simple and efficient mechanism to prevent dangling pointer

propagation.

One desirable but not yet explored feature that pointer neutralization can provide is **object origin tracking (OOT)**. When software crashes due to dangling pointer dereference, OOT can inform us of how a dangling pointer is caused, i.e., where the pointed object is allocated and freed. This information can greatly help programmers pinpoint the root-causes of UaF vulnerabilities. pSweeper achieves OOT by encoding origin information into neutralized dangling pointers. Compared to other user site diagnostic tools that require record-and-replay [209] or multiple failures [130, 148, 149], pSweeper can pinpoint root-causes of UaF vulnerabilities in one safe crash. In particular, it achieves this *at a trivial cost*.

Finally, we implement a prototype of pSweeper and demonstrate its effectiveness using real-world UaF vulnerabilities. Our evaluation results on SPEC CPU2006 benchmarks show that the induced overhead is quite low (9.3%). We demonstrate that pSweeper scales quite well on multi-thread applications using PARSEC benchmarks. We further conduct two case studies with Lighttpd web server and Firefox browser.

6.2 Background and Threat Model

Dangling Pointer. A pointer variable `p` is dangling *iff* an object `O` with address range $\forall$ m, size : `[m, m+size-1]` has been freed and `p`$\in$`[m, m+size-1]`.

In practice, UaF exploits commonly reuse freed memory and fill it with specially crafted contents which are then accessed through dangling pointers. Therefore, it is insufficient to check whether a pointer points to freed memory. Instead, it is imperative to enforce that dangling pointers never point to memory that can be arbitrarily manipulated by attackers.

Threat Model. This paper focuses on UaF vulnerabilities rooted in dangling pointers that can point to any memory region, including heap, stack, code and data. An attacker can crash applications but cannot cause any other consequences. Spatial attacks that exploit out-of-bound writes like buffer overflows, and temporal attacks that exploit uninitialized reads, are out of our scope. Therefore, similar to related

works [139, 212, 225], we do not protect our system from these vulnerabilities and our proposed defense should be used along with orthogonal protectors. We also assume applications do not have concurrency bugs. Finally, we do not deal with undefined behaviors, such as `delete` objects created using `new[]`.

6.3 Overview

6.3.1 High-Level Approach of pSweeper

pSweeper aims to robustly protect against UaF exploits with low overhead and pinpoint the root-causes of UaF vulnerabilities being exploited in the wild. To accomplish these, we propose Concurrent Pointer Sweeping (CPS) and Object Origin Tracking (OOT) in pSweeper. The basic approach of pSweeper follows a similar protection principle to pointer nullification in DANGNULL [139], FreeSentry [225] and DangSan [212]. In particular, when an object is freed, all dangling pointers are neutralized to disrupt UaF exploits. However, pSweeper differs significantly in two key design aspects:

1. How to find dangling pointers; and

2. What value is used to neutralize dangling pointers.

Finding Dangling Pointers. All previous approaches [139, 212, 225] *synchronously* track the pointers that are still pointing to freed objects. This design inevitably incurs undue overhead because it requires range-based queries for pointers.

pSweeper proposes CPS, a totally different design. Our core idea is to *iteratively* sweep all live pointers at runtime in *concurrent* threads to neutralize the dangling pointers. To avoid missing dangling pointers due to memory reuse, CPS delays object frees to the end of every round of sweeping. The main challenge lies in guaranteeing high efficiency and scalability in face of entangled races among CPS and application threads. Ideally, CPS should instrument few code to applications and avoid stalling their threads (i.e., lock-free).

Choosing Value for Pointer Neutralization. Previous works [139, 212, 225] simply set dangling pointers to `NULL` or kernel space. This guarantees that applications crash safely when dangling pointers are accessed.

pSweeper instead specially crafts the values to neutralize dangling pointers. Our key insight is that the crucial information to pinpoint root-causes of UaF vulnerabilities is how a dangling pointer is caused, i.e., how the pointed object is allocated and freed. Therefore, besides enforcing safe crash upon dangling pointer dereference, pSweeper also encodes object origin information into dangling pointers to achieve OOT. Compared with other tools that provide a similar feature to OOT [172, 188], pSweeper is more robust and efficient.

Enforced Protection Protocol. Building upon CPS and OOT, pSweeper will enforce the runtime protection protocol as follows. Given a dangling pointer `p`:

- If `p` is accessed before being neutralized, applications continue to execute *correctly* similar to garbage collection [127].
- If `p` is accessed after being neutralized, applications abort *safely* with object origin information dumped.

6.3.2 An Illustration Example

Figure 6.1 illustrates pSweeper with an example in time line. All `malloc()`, `free()`, and assignment instructions are executed in application threads. R_i and ΔR_i denote the start and end of the i_{th} sweeping round of pSweeper threads, respectively.

Assume the application executes three `malloc()` and one pointer assignment before ΔR_{i-1}. From these instructions, pSweeper identifies four live pointers, `p`, `r`, `q`, and `s` at runtime.

During the interval of sweeping rounds, i.e., between ΔR_{i-1} and R_i, an application thread invokes `free(q)`. However, this free request will be hooked by pSweeper and delayed to the end of i_{th} sweeping round. During the i_{th} sweeping round, pSweeper checks all four pointers to find and neutralize the dangling one q. At ΔR_i, the delayed

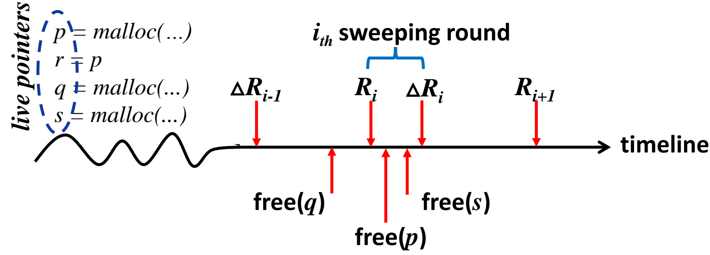


Figure 6.1: Illustration of pSweeper in time line.

`free(q)` gets executed. If two other memory blocks (p and s) are freed during the i_{th} round, they will be delayed to ΔR_{i+1}.

While the overall approach sounds simple, it is non-trivial to efficiently handle the entangled races among pSweeper and application threads. For instance, during the i_{th} sweeping round, assume pSweeper has checked p and r but has not neutralized q. It is possible that an application thread propagates the dangling pointer to a swept one, e.g., executing $r = q$. pSweeper must efficiently handle such cases.

6.3.3 Architecture of pSweeper

To implement CPS and OOT, pSweeper combines compile-time instrumentation and a runtime library, as shown in Figure 6.2. There are three components in pSweeper:

Pointer address identification. pSweeper first statically identifies where pointers will be located at runtime (§6.4.2). It achieves this by analyzing the types of local/global variables. For pointers in dynamically allocated objects, we adopt the same strategy as previous works [139, 212, 225]. Specifically, we rely on the types of operands in `store` instructions. Code is instrumented into applications to bookmark all live pointers.

Concurrent pointer sweeping thread. At runtime, dedicated pSweeper threads iteratively sweep all live pointers and neutralize the dangling ones. The asynchronous nature of CPS requires object frees to be deferred (§6.4.3). Otherwise, when a memory block is freed, it may get reused in applications before CPS threads can neutralize all dangling pointers. The scalability of CPS is guaranteed with two techniques. First, we implement memory allocation status (MAS) table (§6.4.1), a data structure

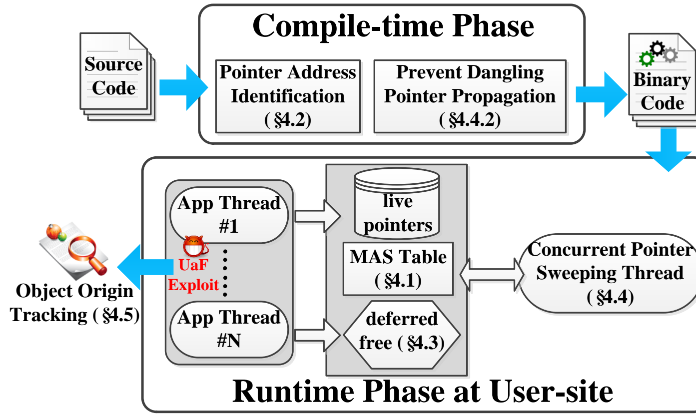


Figure 6.2: The architecture of pSweeper.

similar to shadow memory [168]. With the help of MAS table, CPS threads can decide if a pointer is dangling with one single memory read. Second, we devise a simple and efficient mechanism to prevent dangling pointer propagation (§6.4.4.2) so that dangling pointers are guaranteed to be neutralized in one single round of sweeping.

Object origin tracking (OOT). Finally, pSweeper encodes object origin information into dangling pointers so that once they are dereferenced, pSweeper can inform developers how corresponding objects are allocated and freed (§6.4.5).

6.4 System Design

In this section, we detail the design of pSweeper. Due to the asynchronous design, we need to efficiently handle the entangled races between application and pSweeper threads. In particular, we aim to address these races with lock-free algorithms, which can highly correlate with the memory model of multicore processors. Our current design builds upon a memory model that: (1) loads are not reordered with other loads; (2) stores are not reordered with other stores; and (3) loads may be reordered with older stores to different locations but not with older stores to the same location. This memory model is applied on x86 [22], AMD64, and SPARC[1].

[1] The default mode of SPARC is Total Store Order (TSO).

6.4.1 Memory Allocation Status Table

Since several components of pSweeper rely on efficient check of memory allocation status, we first present the design of MAS table. MAS table is built on the fact that pSweeper only needs to know if a memory address is allocated or freed, and it does not need to know where the object boundaries are. Therefore, similar to the design philosophy in previous works [136, 151, 165, 213, 212], we can simply maintain the memory allocation status in a shadow heap. Upon allocation, all corresponding bytes in shadow heap are set to one and reset to zero upon free. As a result, pSweeper can achieve the check with one single memory read.

However, this naive implementation is still inefficient. First, it incurs high overhead to set and reset shadow heap. Second, it doubles memory consumption. To optimize, we leverage the observation that pragmatic memory allocators usually enforce object size and alignment. For example, the base and size of small and large objects (based on a predefined size threshold) are usually aligned to multiples of the pointer and page size, respectively. Therefore, MAS table only requires 1-byte for every page or 8-byte on x64 (4-byte on x86).

6.4.2 Locating Live Pointers

Pointers can be on stack, data, and heap segments. Dangling pointers on all three regions can be exploited.

6.4.2.1 Pointers on Data Segment

Pointers can reside in data segments, including global and static variables[2]. These pointers can generally be identified at compile time. For each global pointer variable, we instrument a store instruction to log its address to a buffer denoted as `globalptr`. `globalptr` is library-specific, i.e., every library as well as the executable has a dedicated

[2] We use "global variables" for short in the remainder of the paper.

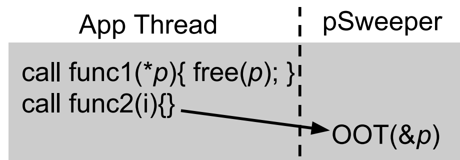


Figure 6.3: Race conditions of pointers on stack.

buffer. pSweeper instruments a `.init` and `.fini` section to every executable and library so that `globalptr` is (de)allocated upon (un)loading.

6.4.2.2 Pointers on Stack

pSweeper handles the pointers in function parameters and local variables in a similar way as global variables. However, due to the asynchronous design of pSweeper, pointers on stack need to be handled specially. Consider the dangling pointer p in Figure 6.3. Before pSweeper neutralizes p, function $func1$ returns and $func2$ is subsequently invoked. Previously storing pointer p, the stack slot now contains a non-pointer variable i. If i by chance has a value equal to the address of a freed memory slot, pSweeper can falsely neutralize it and thus corrupt application data.

To efficiently handle this race, pSweeper relocates all pointers on stack to a dedicated stack denoted as `stackptr`. In this way, every variable in `stackptr` is of pointer type. Thus, pSweeper can safely sweep and neutralize them. However, pointers in complex data types like `struct` and `class` cannot be easily moved to `stackptr` without losing compatibility. We therefore simply allocate all such variables on the heap.

6.4.2.3 Pointers on Heap

Similar to previous works [139, 225, 212], we also rely on the types of operands in `store` instructions to track pointer addresses at runtime. The main difference lies in what task is performed at each pointer `store` instruction. All previous systems require

Algorithm 2 Compile-time store instruction instrumentation for pointer address identification.

```
1: function BOOKMARK_HEAP_POINTER( )
2:     for each storeinst do
3:         if onDataOrStack(storeinst.dest) then
4:             continue
5:         Instrument Bookmark_Ptr() after storeinst.
```

Algorithm 3 Bookmark live pointer addresses.

PtrList: `live pointer list`

```
1: function BOOKMARK_PTR( &ptr )
2:     if notOnHeap(&ptr) then
3:         return
4:     if duplicatePtr(&ptr) then
5:         return
6:     PLM[&ptr] = 1                          ▷ Set pointer mark table.
7:     if objFreed(&ptr) then
8:         PLM[&ptr] = 0
9:         return
10:    appendToList(&ptr, PtrList)
```

to synchronously track *i)* in which object a pointer is located; and *ii)* which object a pointer is pointing to. This inevitably incurs high overhead due to the expensive range-based searches. In contrast, pSweeper simply bookmarks the addresses of live pointers. This, however, is still non-trivial to implement efficiently.

An assignment operation `LHS=RHS` is usually transformed to a compiler intermediate representation (IR) `store <ty> <val>, <ty>* <ptr>`, where `val` is the value in `RHS` and `ptr` is the memory address of `LHS`. If the type `<ty>` of `val` is a pointer, `LHS` is a pointer. However, its address should not be naively bookmarked for three considerations. First, we must ensure the pointer is not on data or stack segments. Second, we should ignore duplicate bookmarks for the same pointer. Finally, the pointer might be in a freed object. Algorithm 2 and 3 show how pSweeper bookmarks live pointers.

Excluding global/local pointers. We exclude global/local pointers in two steps. First, we identify `store` instructions for non-heap pointers at compile time and do not instrument them (Algorithm 2). Second, at runtime, we check if the address of

a pointer is indeed in range of heap (Line 2 Algorithm 3).

Skipping duplicate bookmarks. To achieve this efficiently, we maintain a pointer location mark (PLM) table, a shadow heap similar to MAS table. However, we do not need to compress `PLM` because the incurred overhead is quite low. For every pointer `ptr` on heap, its corresponding slot `PLM[&ptr]`, is set to one. All other bytes in `PLM` are 0. Thus, `duplicatePtr()` can be achieved with a single memory read.

Validity of pointer address. We next check if the object where the pointer is contained has been freed (Line 7 Algorithm 3). This can be efficiently achieved using MAS table. Note that, there is a potential race that the object where `ptr` is contained gets freed and reused by another application thread after the check but before Line 10 Algorithm 3. We discuss this race further in §6.4.4.1.

Live pointer list. We simply use a double-linked list (`PtrList`) to maintain all live pointers. As a result, `appendToList()` is quite efficient. Further, in order to avoid races among application threads which can concurrently operate on `PtrList`, we use a separate list for each thread.

Removing stale pointers. Here we have described how to bookmark live pointers. When an object is freed, all pointers contained in it should be removed from `PtrList`. This is achieved in CPS (§6.4.4.1).

6.4.3 Deferred Free

pSweeper requires object frees to be deferred to the end of a sweeping round. To this end, pSweeper maintains live objects in a double-linked list (`ObjList`) and adds metadata `freeflag` for each object (Figure 6.4). In the hooked `malloc()`, pSweeper first sets `freeflag` to zero (Line 5 Algorithm 4) and then appends the new object to `ObjList` (Line 6 Algorithm 4). When `free()` is invoked in applications, we simply set `freeflag` as in Algorithm 5. Similar to `PtrList`, each application thread uses a thread-local list to maintain objects and nodes in `ObjList` are removed by CPS (§6.4.4.1).

```
struct LiveObjNode{
  obj_addr;   // object address
  freeflag;   // Section §6.4.3
  scanflag;   // Section §6.4.4.1
  slotid;     // Section §6.4.5
  struct LiveObjNode *prev, *next;
};
```

Figure 6.4: Metadata of live objects.

Algorithm 4 Hooked malloc().

1: **function** MALLOC(size)
2: $obj \leftarrow$ **real_malloc**($size$)
3: **setMASTable**(obj)
4: $obj.scanflag \leftarrow 0$
5: $obj.freeflag \leftarrow 0$
6: **appendToObjList**(obj, $ObjList$)
7: **mfence** ▷ Memory barrier

6.4.4 Concurrent Pointer Sweeping (CPS)

CPS is consisted of two components, dedicated CPS threads and dangling pointer propagation instrumentation. Dedicated CPS threads (§6.4.4.1) are the core of CPS and they iteratively sweep live pointers to find and neutralize dangling ones. One challenge here is that application threads can propagate dangling pointers to the pointers that have been neutralized by CPS threads. We devise a simple and efficient mechanism (§6.4.4.2) to prevent dangling pointer propagation in application threads. Next, we describe each component in details. We first assume one CPS thread is spawned for a multi-threaded application and extend to multiple CPS threads in §6.4.4.3.

6.4.4.1 CPS Threads

Algorithm 6 presents the pseudocode of CPS thread whose body is an infinite loop (Line 2) implementing iterative sweeping. CPS takes a list of live objects and pointers as input. In every round of sweeping, CPS threads execute in three steps.

- **Step 1** (Lines 4 $\sim$ 9)

Algorithm 5 Deferred free() invoked in applications.

```
1: function FREE( obj )
2:     assertDoubleFree(obj)                ▷ Abort upon double free.
3:     clearMASTable(obj)
4:     obj.freeflag ← 1
```

This step traverses live object list and if an object's `freeflag` is set, another field of metadata `scanflag` is set. `scanflag` is initialized as 0 in `malloc()` (Line 4 Algorithm 4). `fillWithSlotIndex`() will be described in §6.4.5. This step is required to guarantee that an object whose `freeflag` is set during pointer sweeping is not prematurely freed.

- **Step 2** (Lines 11 ∼ 18)

This step sweeps all live pointers and checks if a pointer is dangling using MAS table (Line 15). Dangling pointers are then neutralized with a value containing object origin information (Line 16). However, this step has a time of check to time of neutralization race as illustrated in Figure 6.5. To be specific, the value of `p` can be modified by application threads after the `isDangling()` check.

To address this, we observe that if a dangling pointer is modified by application threads between `isDangling()` and `OOT()`, we should preserve the value written by application threads and the neutralization by pSweeper can fail safely. On the one hand, if the new value written by application threads points to a live object, the dangling pointer is eliminated by application threads and we must preserve the value for correct execution. On the other hand, if the new value points to a freed object, this propagation will be handled by our mechanism that prevents dangling pointer propagation (§6.4.4.2). Fortunately, modern processors provide efficient hardware instructions such as `lock cmpxchg` that exactly meet our needs.

In addition, CPS threads skip stale pointers, i.e., whose containing objects have been freed, and remove them from `PtrList` (Lines 12∼14). To demonstrate that the race mentioned in §6.4.2.3 does not cause failures in CPS, we consider two cases.

Case 1: Line 7 in Algorithm 3 returns true. In this case, pSweeper always correctly skips stale pointers. In particular, no live pointer is missed when `objFreed()`

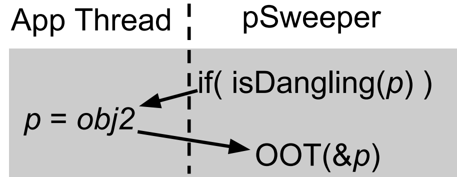


Figure 6.5: Time of check to time of neutralization race.

returns true but the memory has been allocated in a different application thread (due to inconsistency MAS table seen by different cores). This is because the `store` instruction must be executed after the hooked `malloc()` has returned. Otherwise, there is a concurrency bug in applications, which violates our assumptions in §6.2. Line 7 Algorithm 4 enforces that `objFreed()` must return false when the hooked `malloc()` returns.

Case 2: Line 7 in Algorithm 3 returns false. The only problem here lies in the possibility that the object where `ptr` is contained can get freed and reused before Line 10 Algorithm 3. In this case, the stale pointer will be appended to `PtrList`. However, this can happen only if there is a concurrency bug in applications, which violates our assumptions in §6.2.

- **Step 3** (Lines 19 ∼ 25)

CPS threads now traverse object list again to free objects whose `scanflag` is set and remove them from the list. In order to avoid locks between insertion by applications and deletion by CPS, the tail node in `ObjList` is delayed until more nodes have been appended.

Avoiding endless sweeping rounds. Since new objects and pointers are created continuously by application threads, the `while`-loops in the above three steps may not terminate if they are not handled specially. To this end, CPS threads enforce that every round of sweeping terminates at the tail nodes of the lists (Lines 8, 17, and 25) that are recorded at the beginning of the loops (Lines 3 and 10). For Step 1 and 3, this enforcement is required because only objects that have been checked against every live pointer can be safely freed. For Step 2, this strategy is correct and

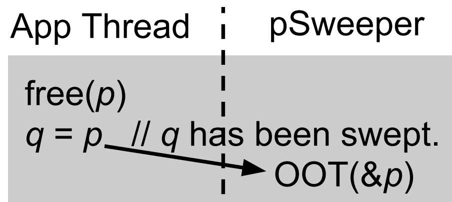


Figure 6.6: Dangling pointer propagation in application threads.

safe because pSweeper prevents dangling pointer propagation (§6.4.4.2) and thus newly added pointers can be deemed as already swept.

6.4.4.2 Preventing Dangling Pointer Propagation

As shown in Algorithm 6, CPS threads sweep every live pointer only once in each round. Unfortunately, dangling pointers can propagate to the swept ones in application threads, as illustrated in Figure 6.6. To handle this race, we consider three ways of pointer propagation: direct assignment (e.g., `q = p`), function arguments (e.g., `func(p)`), and returns (e.g., `p = getPtr()`). Pointers in function arguments are handled in the same way as direct assignments because they have been relocated to `stackptr` (§6.4.2.2) and we currently use direct assignments to initialize stack pointers. Figure 6.7 presents how direct assignment and function return are handled.

Every direct assignment `q = p` is usually compiled to two instructions, one `load` of `p`'s value and one `store` to `q`. To handle this case, we instrument two checks after the `store` instruction. First, we check whether `q` is dangling. If so, we nullify it. Then, we check whether `p`'s value has been neutralized. If so, we store the new value to `q`. We prove correctness of this mechanism as follows:

- **Precondition.** Since we assume no concurrency bug, no one else except CPS thread will modify `p` or `q` during the code sequence in Figure 6.7.
- **Fact.** The race is harmful *iff* `q` is swept before `p`.
- **Completeness.** To prove the completeness of this mechanism, we only need to prove that, if both checks fail, `q` must NOT be dangling. We use proof by

```
%r1 = load  p
store %r1, q
%r2 = load q
if ( isDangling(%r2) )
   OOT(&q)
%r3 = load p
if (%r3 != %r1)
   store %r3, q
```

q = p

Figure 6.7: Prevent dangling pointer propagation. Code snippets with a dark background are instrumented by pSweeper.

contradiction. **Proof:** Assume (`q` is dangling) $\implies$ (`p` has not been neutralized before `%r3=load p`) $\implies$ (the pointed memory is still freed before `%r3=load p`) $\implies$ (`isDangling(%r2)` must return true) $\implies$ (`q` is set to NULL and `q` is not dangling). This contradicts the initial assumption.

- **Soundness.** We need to prove that, if either check succeeds, `q` must be dangling. The proof is straightforward based on the two preconditions.

Note that, we must insert `__asm__ __volatile__("":::"memory")` between the `load` instructions to prevent reordering by compilers. But, we do not need to insert memory barriers before `%r3=load p` because we only need to ensure that this `load` happens after the one in `isDangling(%r2)` but do not care if `store` instructions have been globally visible before `%r3=load p`.

6.4.4.3 More pSweeper Threads

pSweeper currently uses only one thread, which is sufficient in our evaluations. However, it can be extended to use multiple threads. The live pointers can be partitioned to segments, with each one being handled by one pSweeper thread during every round

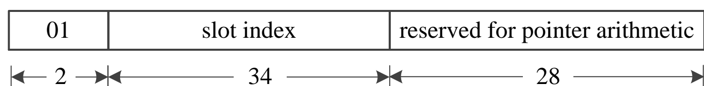


Figure 6.8: Use of pointer bits by OOT.

of sweeping. In this extension scheme, there is no race among pSweeper threads, and thus, no synchronization is required, making pSweeper quite scalable.

6.4.5 Object Origin Tracking (OOT)

It is notoriously difficult to analyze and locate bugs triggered in production runs [130, 148, 149, 209]. In order to facilitate the root-cause diagnosis of UaF vulnerabilities, pSweeper aims to provide not only where dangling pointers are dereferenced (which can be obtained in core dumps) but also how objects are allocated and freed, i.e., object origin tracking (OOT). Unfortunately, it is non-trivial to link a dangling pointer access to the corresponding improper memory (de)allocation. Existing approaches like AddressSanitizer [188] and Exterminator [172] bind origin information with objects. However, this can cause inaccurate OOT when memory is reused, which is common in UaF exploits. Therefore, they are primarily suitable for in-house debugging but not in-production diagnosis.

pSweeper instead encodes object origin information into dangling pointers. Such information is independent to memory reuse and can be propagated at no extra cost. Moreover, The most significant two bits are set to `01` as in Figure 6.8 to make sure that applications crash safely upon dangling pointer dereference. Then, the origin information can be obtained in signal handlers. However, we must reserve sufficient least-significant bits to support pointer arithmetics. This makes it difficult to achieve OOT on 32-bit platforms. As a result, pSweeper currently supports x86-64 only.

OOT records the call stacks of `malloc()` and `free()` in a buffer slot which is assigned an index. The index is encoded into the middle 34 bits during pointer neutralization, as shown in Figure 6.8. To reduce the memory overhead, the call stack information is compressed. In order to retrieve the slot index in `OOT`, pSweeper fills freed objects with corresponding slot indexes as in Line 7 Algorithm 6. In this way,

given an in-bounds dangling pointer `p` to an object, pSweeper can easily construct the value to neutralize `p`.

When applications crash due to dangling pointer dereference, pSweeper extracts OOT information in signal handlers. However, Linux always returns zero, instead of the tagged pointer in Figure 6.8, as the illegal address in signal handlers. We address this by first obtaining the faulty instruction, e.g., `4008fe: movl %edx, (%rax)`, through `EIP/RIP` in signal handlers. This instruction informs that register `rax` contains the pointer value. Then, we can obtain the encoded origin information by reading that register.

6.5 Evaluation

We implement a pSweeper prototype for x86-64, on top of LLVM 3.7 compiler infrastructure [27], and use LLVM's link-time optimization support (LTO) for the whole program analysis. The static analysis and instrumentation pass in pSweeper operates on LLVM intermediate representation (IR). Our current prototype employs some preliminary optimizations, e.g., inlining operations in Algorithm 3 and Figure 6.7 when instrumenting `store` instructions to avoid function calls.

We evaluate pSweeper by answering four questions:

- Is pSweeper effective to mitigate real UaF vulnerabilities?
- What is the performance overhead of pSweeper?
- How scalable is pSweeper for multi-threaded applications?
- Can pSweeper efficiently work on complex software?

All experiments are conducted on 64-bit Ubuntu-14.04 with a 2-core 4-thread Intel i5-4300U at 1.9GHz with 12GB RAM.

6.5.1 Effectiveness of pSweeper

To evaluate the effectiveness of pSweeper, we apply it to four real-world UaF vulnerabilities in three applications, as listed in Table 6.1. pSweeper successfully neutralizes the unsafe dangling pointers and pinpoints the root-causes in all four cases. Due to space limit, we next describe CVE-2016-6309 only in details.

CVE/Bug ID	Application	Protected
CVE-2016-6309 [11]	OpenSSL 1.1.0a	✔
CVE-2014-3505 [10]	OpenSSL <1.01i	✔
Bug 12840 [42]	Wireshark	✔
Bug 2440 [26]	Lighttpd 1.4.32	✔

Table 6.1: Real-world UaF vulnerabilities used for evaluation.

CVE-2016-6309 in OpenSSL is caused by memory reallocation in `statem.c:548`. OpenSSL initially allocates a buffer of 16KB to receive messages. When a larger message is received, the buffer is reallocated using `CRYPTO_clear_realloc()`, which essentially allocates a new buffer and frees the old one. Therefore, the underlying location of the buffer is changed. However, a pointer `s→init_msg` is not updated and still refers to the old location.

When this vulnerability is exploited, there can be two cases. First, due to deferred free and asynchronous neutralization, if the dangling is accessed before being neutralized, the openSSL server can always execute normally. On the other hand, if it is exploited after neutralization, the openSSL server crashes safely and pSweeper successfully pinpoints `OPENSSL_clear_realloc()` in `BUF_MEM_grow_clean()` (`buffer.c:109`) as root cause.

6.5.2 Performance on SPEC CPU2006

We next evaluate the performance overhead of pSweeper on SPEC CPU2006 benchmarks. Unfortunately, we cannot run benchmarks dealII, omnetpp, and xalancbmk because our baseline LLVM fails to compile, giving errors like `non-constant-expression`

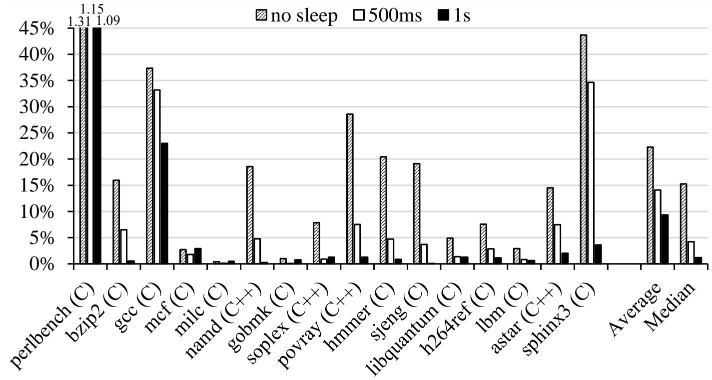


Figure 6.9: pSweeper's performance on SPEC CPU2006.

`cannot be narrowed from type 'XMLInt32'`". Table 6.2 presents the statistical results of SPEC CPU2006 benchmarks when pSweeper runs at a sweeping rate of one second.

As can be seen, pSweeper finds similar number of pointers (Column 5 in Table 6.2) as DangSan, which is far more than DANGNULL. This demonstrates that pSweeper has comparative coverage to the state-of-the-art defense systems. We also find that pSweeper neutralizes fewer pointers (Column 7 in Table 6.2) than DangSan, although more than DANGNULL. This is because pSweeper concurrently sweeps dangling pointers in a dedicated thread and does not stall applications. As a result, although a pointer is dangling at `free()`, it probably has been overwritten by applications with non-dangling values when pSweeper checks it. In particular, the majority of dangling pointers identified in DangSan are on stack [212], which become invalid after function returns. Also, it is possible that the objects containing dangling pointers have been freed before pSweeper starts to sweep. All of these invalid dangling pointers are ignored by pSweeper. We emphasize that neutralizing these stale dangling pointers does not increase the security guarantee and pSweeper provides the same protection as previous systems.

Benchmark	# of Allocations	# of Frees	Avg. Object Size (Bytes)	Total # of Pointers	Peak # of Pointers	# of Pointers Neutralized
perlbench (C)	358M†	356M	514	40,490M	971,353	421,352
bzip2 (C)	7,182	4,440	1.6M	2.2M	1,184	0
gcc (C)	28M	28M	26,088	7,170M	438,016	186,451
mcf (C)	1,174	721	1.4M	7,658M	173,625	0
milc (C)	7,686	7,184	11M	2,585M	76,254	0
namd (C++)	2,493	2,038	19,582	2.9M	1,746	0
gobmk (C)	663,879	658,695	1,707	607M	28,841	86
soplex (C++)	312,951	310,613	189,852	836M	76,278	553
povray (C++)	2.4M	2.4M	56	4,679M	128,525	7,428
hmmer (C)	2.4M	2.4M	1,048	3.8M	2,237	0
sjeng (C)	1,174	717	154,809	3	0	0
libquantum (C)	1,348	895	1.1M	186	8	0
h264ref (C)	182,784	181,283	7,735	11M	3,674	961
lbm (C)	1,173	720	367,047	5,949	28	0
astar (C++)	4.8M	4.8M	922	1,235M	34,519	104
sphinx3 (C)	14M	14M	1,136	302M	24,923	762

Table 6.2: Detailed results on SPEC CPU2006 benchmarks. pSweeper runs at 1s sweeping rate. †M for million.

6.5.2.1 Runtime Overhead

Figure 6.9 presents the performance overhead of pSweeper at different sweeping rates, i.e., no sleep, 500ms sleep, and 1s sleep between sweeping rounds. The overhead is normalized over the baseline and all the results are averaged over three consecutive runs. The average overheads of pSweeper at different sweeping rates are 22.2% (no sleep), 14.1% (500ms), and 9.3% (1s).

Effect of sweeping rate. Theoretically, faster sweeping rates should not significantly affect the performance of applications as pSweeper concurrently runs on spare cores. However, we find that in the case of no sleep, most of the overhead comes from the interference of full-speed pSweeper threads. In particular, about 15.7% of the averaged overhead and 14.2% of `gcc` in this setting can be attributed to this factor. In order to isolate the potential sources of interference, we turn the pSweeper thread to a simple empty loop, i.e., `while(1){}` and instantly free memory in `free()` hook.

It causes similar overhead. Therefore, we derive that this is not due to the factors like cache interference. Instead, it is caused by inefficient performance isolation among threads in the Linux kernel, and we are investigating this problem in kernel.

When pSweeper thread runs at a slower speed like 1s, it induces trivial overhead on almost all benchmarks except `perlbench` and `gcc`. `perlbench` still suffers high overhead due to the exceptionally large volume of object allocations. Simply intercepting `malloc()` with `LD_PRELOAD` can incur about 8% overhead. For `gcc`, although the interference from pSweeper thread is minimized, much more time is spent in kernel mode when allocating memory. This is because all frees are deferred to the end of a sweeping round. An allocation-intensive application like `gcc` may not be able to immediately reuse the freed memory. As a result, memory allocators need more time to allocate a new object.

Static instrumentation overhead. We now break down the overhead caused by static code instrumentation. The overhead mainly comes from the hooked `malloc()` family of functions, which set up object metadata, maintain live objects and MAS table. They introduce a bunch of extra memory writes for each allocated object. We find that they account for about 5.6% of the average overhead. Especially, in the case of allocation-intensive applications, the accumulated overhead is high, e.g., ∼15% for `gcc`. Another important source of overhead comes from recording object origin information. In particular, we need to track the functions call sequences to `malloc()`-family of functions. Although we have made some optimizations, e.g., recording the hash value of function names instead of strings, the overhead is still about 1.3%. Finally, the instrumented `store` instruction, which is the main performance bottleneck in previous works [139, 225, 212], causes low overhead in pSweeper, about 1.8%.

Dynamic instruction count and data cache overhead. We use hardware performance counters to measure the dynamic instruction counts and cache misses of the 32KB L1 data cache. We are only interested in the overhead caused by the instrumented code. Thus, we do not spawn the pSweeper thread and disable deferred free. The results are plotted in Figure 6.10, showing that dynamic instruction counts highly

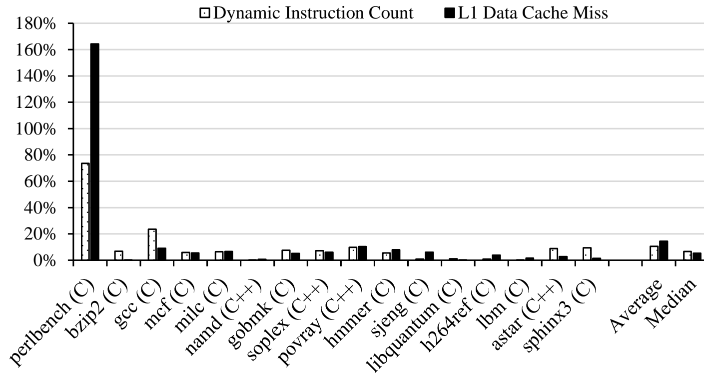


Figure 6.10: Dynamic instruction overhead and L1 data cache misses on SPEC CPU2006.

correlate with the runtime overhead and are the main source of overheads for most benchmarks. A noticeable negative impact of MAS and PLM tables is the additional data cache misses, resulting in a large portion of performance overhead.

6.5.2.2 Memory Overhead

Figure 6.11 shows that pSweeper moderately increases memory footprint in terms of maximum resident set size, with average overheads 44.2% (no sleep), 60.8% (500ms), and 128% (1s).

Generally, faster sweeping rates result in lower memory overhead. This is because free requests are deferred shorter, and thus memory can be freed faster. A faster sweeping rate is especially important to allocation-intensive applications. For instance, sweeping at 500ms, compared to 1s, reduces the memory overhead of `gcc` by an order of magnitude. Other sources of memory overhead include MAS table, `ObjList`, `PtrList` and `PLM` table. We can see that these metadata consumes acceptable amount of memory. In particular, since several benchmarks allocate a small number of large objects (Column 4 Table 6.2), the compression strategy used in MAS table can greatly reduce memory overhead.

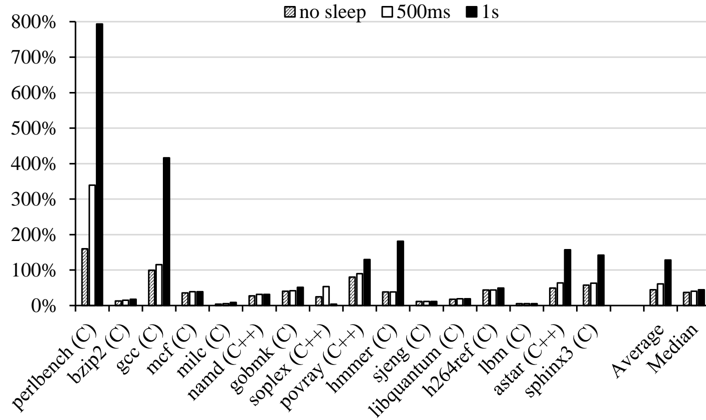


Figure 6.11: Memory overhead on SPEC CPU2006.

6.5.2.3 Comparison to DangSan

We compare pSweeper with DangSan, a state-of-the-art UaF defense system outperforming all existing systems like DANGNULL and Oscar [93]. Since some benchmarks fail to compile in our evaluation, we compare over intersection of the benchmarks between pSweeper and DangSan. pSweeper runs at 1s sweeping rate. The geometric mean of DangSan is a 38% slowdown, while ours is 9.3%. Figure 6.12 compares the seven benchmarks on which pSweeper obviously outperforms DangSan. There is no remarkable difference on other benchmarks. In terms of memory overhead, DangSan imposes an average overhead of 210%, while ours is 128%.

6.5.3 Scalability on Multi-threaded Applications

We use PARSEC 3.0 [73] to evaluate the scalability of pSweeper with respect to an increasing number of application threads. Again, our baseline LLVM fails to compile four benchmarks and Figure 6.13 shows the results for nine succeeded ones. As we can see, pSweeper scales nearly as well as the baseline on all benchmarks. This is mainly because lock-free algorithms are devised to address almost all races between pSweeper and application threads. As a result, the incurred overhead does not increase significantly when systems become more contended. We observe that the

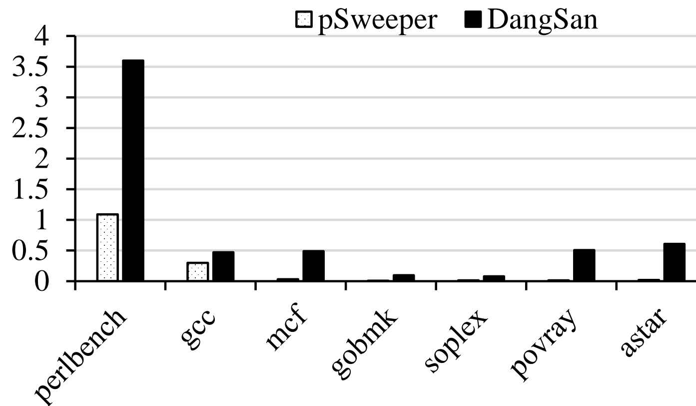


Figure 6.12: Overhead comparison with DangSan.

runtime of benchmarks does not decrease anymore after the number of application threads reaches four. This is because our CPU has only two cores providing 4-thread hyperthreading. The geometric means of overhead over all nine benchmarks range from 7.5% to 12.9% for pSweeper-1s, 8.9% to 14.4% for pSweeper-500ms, and 21.1% to 32.8% for pSweeper-nosleep. The memory overhead basically does not highly correlate with the number of application threads and the geometric means of overheads are 1600% (1s), 840% (500ms), and 49% (no sleep), respectively. The high overhead is mostly due to `swaptions` that consumes 144x memory for pSweeper (1s). The reason is that the memory footprint of baseline `swaptions` is quite small. As a result, the memory consumption caused by pSweeper becomes exceptionally large relative to the baseline. Excluding `swaptions` that is not evaluated by DangSan, the overheads of pSweeper become 27% (1s), 22% (500ms), and 18% (no sleep), respectively.

6.5.4 Macro Benchmarks

We now demonstrate that pSweeper works efficiently on modern applications with two case studies, Lighttpd web server and Firefox browser.

6.5.4.1 Lighttpd

We first conduct experiments on Lighttpd 1.4.40. To generate client requests, we run

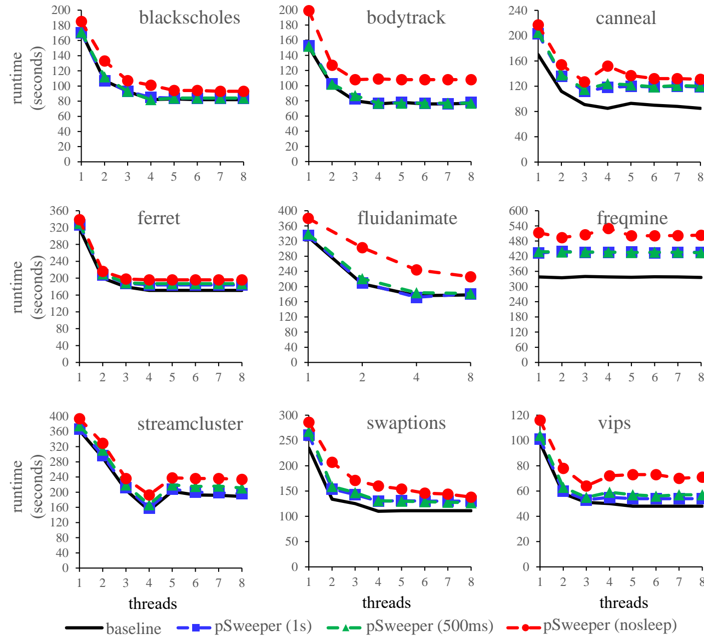


Figure 6.13: Scalability of pSweeper on PARSEC 3.0. The number of threads must be a power of two for `fluidanimate`.

the ApacheBench [4] tool on a second desktop. The tool makes 100,000 requests with 128 concurrent connections to transfer a 50-byte file. We use a very small file to minimize the potential variance caused by network and I/O. The results are averaged over five runs.

Figure 6.14 shows the throughput of Lighttpd with respect to the different number of worker processes. We can see that pSweeper scales well on Lighttpd with overheads in ranges of 3.7%∼13.6% (1s), 4.1%∼15.9% (500ms), and 8%∼22.8% (no sleep). The memory overheads are about 158.7% (1s), 42.2% (500ms), and 38.7% (no sleep) in all cases.

6.5.4.2 Mozilla Firefox

We choose Firefox 47.0 as our second case study. Table 6.3 presents the evaluation results on three popular browser benchmarks, MotionMark [32] assessing a browsers

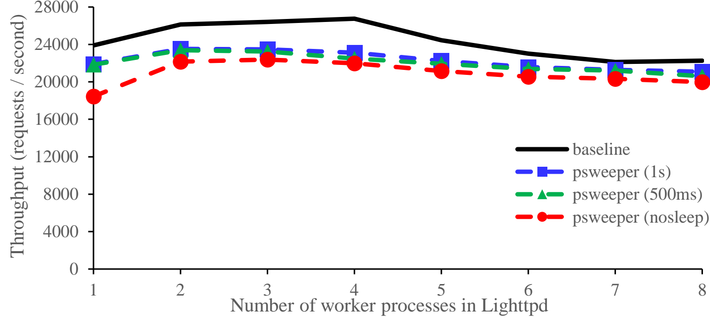


Figure 6.14: pSweeper overhead on Lighttpd.

capability to animate complex scenes at a target frame rate, Speedometer [39] measuring simulated user interactions in web applications, and JetStream 1.1 [23] covering a variety of advanced Javascript workloads. In all three benchmarks, larger scores indicate higher performance. We can see that the induced runtime overhead is quite low, ranging from 2.3% to 7.7% for pSweeper-1s. The geometric means of memory overhead are 863%, 374%, and 117%, for pSweeper-1s, -500ms, and -nosleep, respectively.

We further evaluate pSweeper by visiting Alexa Top 50 websites. We encounter no error during the test of accessing the websites. Table 6.4 lists the page load time (using app.telemetry [5]) when visiting five popular websites. On average, the page load time increases by 3.1% , 7.6%, and 18.6% for pSweeper-1s, -500ms, and -nosleep, respectively.

Benchmarks	**MotionMark**	**Speedometer**	**JetStream 1.1**
	Score	**Runs /minute**	**Score**
Baseline	145.24	48.5	160.63
1s	133.98 (7.7%)	44.4 (6.9%)	156.9 (2.3%)
500ms	130.77 (9.9%)	41.47 (13%)	156.41 (2.6%)
nosleep	107.4 (26%)	36.9 (22.6%)	132.03 (17.8%)

Table 6.3: Overhead of pSweeper(-1s, -500ms, -nosleep) on three browser benchmarks. The percentage in parentheses is the slowdown.

6.6 Discussion & Limitations

pSweeper metadata protection. pSweeper does not specially protect its metadata like the MAS and PLM table. However, this does not degrade our security guarantee. By design, all UaF exploits are disrupted. Thus, attackers can leak and

Websites	Baseline	pSweeper		
		1s	500ms	nosleep
google.com	0.55	0.57	0.59	0.62
youtube.com	1.95	1.99	2.08	2.18
facebook.com	0.64	0.66	0.73	0.83
amazon.com	2.42	2.51	2.52	2.95
yahoo.com	3.51	3.61	3.66	4.12

Table 6.4: Page load time (in seconds) of pSweeper on five popular websites.

tamper with metadata only through non-UaF vulnerabilities. As discussed in §6.2, orthogonal defenses should be used to protect against these vulnerabilities.

Accessing freed memory due to deferred free. Since pSweeper defers object free until the end of a sweeping round, applications are able to access the memory that should have been freed. This design resembles garbage collection. Therefore, we believe this is not a critical concern in practice.

Energy consumption. Since pSweeper continuously scans for dangling pointers in a concurrent thread, it will consume more power and energy. As a result, it may not be suitable for deployment on battery-backed mobile devices. Instead, we envision pSweeper to be mainly deployed on desktops.

False positives. Basically, false positives can occur in two cases. First, a pointer may be type-casted to and used as an integer. For instance, a program might depend on the difference of two pointers p, q. If p or q is neutralized by pSweeper, the value $(p - q)$ will be changed. Second, applications may intentionally use the values in dangling pointers.Since these false positives are rare in practice, we believe they will not seriously affect the practicality of pSweeper. Actually, all existing approaches [139, 225, 212] suffer the same false positives.

False negatives. pSweeper relies on the types of global/local variables and operands in `store` instructions to identify live pointers. However, an integer is type-casted to a pointer at runtime. Also, pSweeper currently conservatively ignores unions if one of their fields are non-pointers. In these cases, pSweeper will suffer false negatives if the missed pointers become dangling.

Another possible cause of false negatives lies in the fact that pSweeper does not proactively neutralize dangling pointers in registers. It will induce undue overhead if pSweeper peeks into and tampers with the registers used by application threads. While these dangling pointers are theoretically false negatives, they can hardly be exploited in practice. Therefore, currently we do not tackle them. Instead, we guarantee that they never propagate to memory (§6.4.4.2). Again, all existing approaches [139, 225, 212] do no handle dangling pointer in registers.

6.7 Conclusion

This paper presents pSweeper, a system that effectively protects applications from UaF vulnerabilities at low overhead. The key feature of pSweeper is to iteratively sweep live pointers to neutralize dangling ones in concurrent threads. To accomplish this, we devise lock-free algorithms to address the entangled races among pSweeper and application threads, without using any heavyweight synchronization mechanism that can stall application threads. We also propose to encode object origin information into dangling pointers to achieve object origin tracking, which helps to pinpoint the root-causes of UaF vulnerabilities. We implement a prototype of pSweeper and validate its effectiveness and efficiency in production environments.

Algorithm 6 Concurrent Pointer Sweeping (CPS) threads.

ObjList: `live object list`
PtrList: `live pointer list`

```
1:  function CPS_Thread( )
2:      while True do
3:          objEnd ← getObjectListTail(ObjList)
4:          while obj ← getNextObj(ObjList) do
5:              if obj.freeflag then
6:                  obj.scanflag ← 1
7:                  fillWithSlotIndex(obj, obj.slotid)
8:              if obj == objEnd then
9:                  break
10:         ptrEnd ← getPtrListTail(PtrList)
11:         while ptr ← getNextPtr(PtrList) do
12:             if objFreed(&ptr) then
13:                 removePtr(&ptr, PtrList)
14:                 continue
15:             if isDangling(ptr) then
16:                 OOT(&ptr)
17:             if ptr == ptrEnd then
18:                 break
19:         while obj ← getNextObj(ObjList) do
20:             if obj.scanflag then
21:                 real_free(obj)
22:                 removeObj(ObjList, obj)
23:                 clearPLMTable(obj)
24:             if obj == objEnd then
25:                 break
26:         Sleep(t)                                    ▷ Decide sweeping rate
```

Chapter 7

CONCLUSION

In this dissertation, we present solutions to address four newly emerging threats, malicious PDF documents, dangling DNS records, domain shadowing, and UaF exploits.

First, we develop a novel malicious PDF detector, which leverages five new static features and the context-aware behavior monitoring. The static features characterize the obfuscation techniques that are widely used by malicious PDF. The context-aware monitoring is based on the observation that the indispensable operations for malicious Javascript to compromise target systems rarely occur in JS-context. Our extensive evaluations on 18,623 benign and 7,370 malicious samples demonstrate that our approach can accurately detect and confine malicious Javascript in PDF with minor performance overhead.

Second, we present the first study to understand and detect a newly emerging threat in domain name system (DNS), called domain shadowing, where miscreants compromise legitimate domains and spawn malicious subdomains under them. Bootstrapped with a set of manually confirmed shadowed domains, we identify a set of novel features that uniquely characterize domain shadowing by analyzing the deviation from their apex domains and the correlation among different apex domains. Building upon these features, we train a classifier and apply it to detect shadowed domains on the daily feeds of VirusTotal, a large open security scanning service. Our study highlights domain shadowing as an increasingly rampant threat since 2014. Moreover, we reveal that they are also widely exploited for phishing attacks. Finally, we observe that several domain shadowing cases exploit the wildcard DNS records, instead of algorithmically generating subdomain names.

Third, we unveil a largely overlooked threat in DNS: a dangling DNS record (Dare), which could be easily exploited for domain hijacking. Specifically, we have presented three attack vectors, IP in cloud, abandoned third-party services, and expired domains. We have quantified the magnitude of Dares in the wild through a large-scale measurement study, during which hundreds of Dares are found on even those well-managed zones like `edu` and Alexa top 10,000 websites. To this end, we have proposed three defense mechanisms that can effectively mitigate Dares with minor human effort.

Finally, we present pSweeper, a defense system that effectively protects applications from UaF vulnerabilities at low overhead. To accomplish our design goals, we propose two unique and innovative techniques, concurrent pointer sweeping (CPS) and object origin tracking (OOT). We have demonstrated the effectiveness of pSweeper using real-world UaF vulnerabilities. Our evaluation results on SPEC CPU2006 and PARSEC benchmarks show that the induced overhead is quite low and pSweeper scales quite well on multi-thread applications. We have further conducted two case studies with Lighttpd web server and Firefox browser to validate the efficacy of eSweeper.

BIBLIOGRAPHY

[1] Aliyun. https://www.aliyun.com/.

[2] Amazon EC2. https://aws.amazon.com/ec2/.

[3] Amazon EC2 and Amazon virtual private cloud. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-vpc.html.

[4] Apachebench. https://httpd.apache.org/docs/2.4/programs/ab.html/.

[5] app.telemetry page speed monitor. https://addons.mozilla.org/en-US/firefox/addon/apptelemetry/.

[6] AWS IP address ranges. http://docs.aws.amazon.com/general/latest/gr/aws-ip-ranges.html.

[7] Azure SDK for python. http://azure-sdk-for-python.readthedocs.io/en/latest/.

[8] Boto: A python interface to amazon web services. http://boto.cloudhackers.com/en/latest/.

[9] CrunchBase. https://www.crunchbase.com/.

[10] Cve-2014-3505. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-3505.

[11] Cve-2016-6309. https://bugzilla.redhat.com/show_bug.cgi?id=CVE-2016-6309.

[12] Detecting memory access errors using hardware support. https://blogs.oracle.com/raj/entry/detecting_memory_access_errors_using.

[13] Doppelganger domain. https://en.wikipedia.org/wiki/Doppelganger_domain.

[14] Fishing the AWS IP pool for dangling domains. http://www.bishopfox.com/blog/2015/10/fishing-the-aws-ip-pool-for-dangling-domains/.

[15] Free pdf password remover. http://www.4dots-software.com/pdf-utilities/free-pdf-password-remover/.

[16] A garbage collector for c and c++. http://hboehm.info/gc/.

[17] GeoIQ. https://crunchbase.com/organization/fortiusone.

[18] GoDaddy. https://www.godaddy.com/.

[19] Google Apps. https://support.google.com/a/answer/112038?hl=en.

[20] Hostile subdomain takeover. http://labsdetectify.wpengine.com/2014/10/21/hostile-subdomain-takeover-using-herokugithubdesk-more/.

[21] Implicit MX. http://tools.ietf.org/html/rfc5321#section-5.

[22] Intel 64 and ia-32 architectures software developer manual. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[23] Jetstream. http://browserbench.org/JetStream/.

[24] Let's Encrypt. https://letsencrypt.org/.

[25] libemu. http://libemu.carnivore.it/.

[26] Lighttpd bug-2440. https://redmine.lighttpd.net/issues/2440.

[27] The llvm compiler infrastructure. http://llvm.org/.

[28] Loaddllviaappinit. http://blog.didierstevens.com/2009/12/23/loaddllviaappinit/.

[29] Mailgun. https://www.mailgun.com/.

[30] Microsoft Azure. https://azure.microsoft.com/en-us/.

[31] Microsoft Azure datacenter ip ranges. https://www.microsoft.com/en-us/download/details.aspx?id=41653.

[32] Motionmark. http://browserbench.org/MotionMark/.

[33] Rackspace. https://www.rackspace.com/en-us.

[34] Regions and availability zones. http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/using-regions-availability-zones.html.

[35] The rise in the exploitation of old pdf vulnerabilities. http://blogs.technet.com/b/mmpc/archive/2013/04/29/the-rise-in-the-exploitation-of-old-pdf-vulnerabilities.aspx.

[36] Sandboxie. http://www.sandboxie.com/.

[37] Selenium. http://www.seleniumhq.org/.

[38] Shopify. https://www.shopify.com/.

[39] Speedometer. http://browserbench.org/Speedometer/.

[40] SSL test. https://www.ssllabs.com/ssltest/analyze.html?d=can.travelocity.com.

[41] Wepawet. http://wepawet.cs.ucsb.edu/.

[42] Wireshark bug-12840. https://bugs.wireshark.org/bugzilla/show_bug.cgi?id=12840.

[43] Working with the appinit_dlls registry value. http://support.microsoft.com/kb/197571.

[44] Your default VPC and subnets. http://docs.aws.amazon.com/AmazonVPC/latest/UserGuide/default-vpc.html.

[45] An invariant form for the prior probability in estimation problems. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 1946.

[46] *PDF Reference (sixth edition)*, 2006.

[47] *JavaScript for Acrobat API Reference*, 2007.

[48] Making malicious pdf undetectable. http://www.signal11.eu/en/research/articles/malicious_pdf.html, 2009.

[49] Another nasty trick in malicious pdf. https://blog.avast.com/2011/04/22/another-nasty-trick-in-malicious-pdf/, 2011.

[50] The number of the beast. http://vinsula.com/cve-2013-0640-adobe-pdf-zero-day-malware/, 2013.

[51] Protected mode. http://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/protectedmode.html, 2013.

[52] Cve-2009-1492. http://www.cvedetails.com/cve/CVE-2009-1492/, accessed in August 2013.

[53] Add javascript to existing pdf files (python). http://blog.rsmoorthy.net/2012/01/add-javascript-to-existing-pdf-files.html, accessed in June 2013.

[54] Contagiodump collection. http://contagiodump.blogspot.com/, accessed in June 2013.

[55] http://www.cvedetails.com/product/497/adobe-acrobat-reader.html?vendor_id=53, accessed in June 2013.

[56] *PROCESS_MEMORY_COUNTERS_EX structure*, accessed in June 2013.

[57] Domain Shadowing With a Twist. https://blog.malwarebytes.com/threat-analysis/2015/04/domain-shadowing-with-a-twist/. 2015.

[58] Pieter Agten, Wouter Joosen, Frank Piessens, and Nick Nikiforakis. Seven months worth of mistakes: A longitudinal study of typosquatting abuse. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[59] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *Proceedings of the 19th USENIX Conference on Security (Security)*, 2010.

[60] Sumayah Alrwais, Xiaojing Liao, Xianghang Mi, Peng Wang, Xiaofeng Wang, Feng Qian, Raheem Beyah, and Damon McCoy. Under the shadow of sunshine: Understanding and detecting bulletproof hosting on legitimate service provider networks. In *IEEE S&P*, 2017.

[61] Sumayah Alrwais, Kan Yuan, Eihal Alowaisheq, Zhou Li, and XiaoFeng Wang. Understanding the dark side of domain parking. In *USENIX Security Symposium (USENIX Security)*, 2014.

[62] Sumayah Alrwais, Kan Yuan, Eihal Alowaisheq, Xiaojing Liao, Alina Oprea, XiaoFeng Wang, and Zhou Li. Catching predators at watering holes: Finding and understanding strategically compromised websites. In *Proceedings of the 32Nd Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[63] David S. Anderson, Chris Fleizach, Stefan Savage, and Geoffrey M. Voelker. Spamscatter: Characterizing internet scam hosting infrastructure. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, 2007.

[64] Fake Extensions Angler EK: More Obfuscation and Other Nonsense. http://blogs.cisco.com/security/talos/angler-update. 2015.

[65] Manos Antonakakis, Roberto Perdisci, David Dagon, Wenke Lee, and Nick Feamster. Building a dynamic reputation system for DNS. In *USENIX Security*, 2010.

[66] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou, II, and David Dagon. Detecting malware domains at the upper dns hierarchy. In *Proceedings of the 20th USENIX Conference on Security*, 2011.

[67] Manos Antonakakis, Roberto Perdisci, Yacin Nadji, Nikolaos Vasiloglou, Saeed Abu-Nimeh, Wenke Lee, and David Dagon. From throw-away traffic to bots: Detecting the rise of dga-based malware. In *USENIX Security*, 2012.

[68] Internet Archive. https://archive.org/. 2017.

[69] R. Arends, R. Austein, M. Larson, D. Massey, and S. Rose. DNS security introduction and requirements. In *RFC 4033 (Proposed Standard)*, 2005.

[70] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI)*, 1994.

[71] Steven M. Bellovin. Using the domain name system for system break-ins. In *USENIX Security*, 1995.

[72] Emery D. Berger and Benjamin G. Zorn. Diehard: Probabilistic memory safety for unsafe languages. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2006.

[73] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[74] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi. EXPOSURE: finding malicious domains using passive DNS analysis. In *NDSS*, 2011.

[75] Website blocked as malicious. https://forum.avast.com/index.php?topic=167705.0/. 2015.

[76] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Softw. Pract. Exper.*, 18, 1988.

[77] Kevin Borgolte, Christopher Kruegel, and Giovanni Vigna. Delta: Automatic identification of unknown web-based infection campaigns. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[78] Leo Breiman and Adele Cutler. Random forests. In *https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm*, 2017.

[79] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees.* CRC press, 1984.

[80] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, 2012.

[81] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel. Prophiler: A fast filter for the large-scale detection of malicious web pages. In *Proceedings of the 20th International Conference on World Wide Web (WWW)*, 2011.

[82] Sundown EK: You Better Take Care. http://blog.talosintelligence.com/2016/10/sundown-ek.html. 2016.

[83] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2015.

[84] Xin Chen, Haining Wang, Shansi Ren, and Xiaodong Zhang. DNScup: Strong cache consistency protocol for DNS. In *IEEE ICDCS*, 2006.

[85] CommonCrawl. http://commoncrawl.org/. 2017.

[86] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. Ccured in the real world. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[87] Marco Cova, Christopher Kruegel, and Giovanni Vigna. Detection and analysis of drive-by-download attacks and malicious javascript code. In *Proceedings of International conference on World wide web (WWW)*, 2010.

[88] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[89] Charlie Curtsinger, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. Zozzle: fast and precise in-browser javascript malware detection. In *Proceedings of USENIX Security Symposium*, 2011.

[90] David Dagon, Manos Antonakakis, Kevin Day, Xiapu Luo, Christopher P. Lee, and Wenke Lee. Recursive DNS architectures and vulnerability implications. In *NDSS*, 2009.

[91] David Dagon, Manos Antonakakis, Paul Vixie, Tatuya Jinmei, and Wenke Lee. Increased DNS forgery resistance through 0x20-bit encoding: Security via leet queries. In *ACM CCS*, 2008.

[92] David Dagon, Chris Lee, Wenke Lee, and Niels Provos. Corrupted DNS resolution paths: The rise of a malicious resolution authority. In *NDSS*, 2008.

[93] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th USENIX Security Symposium*, 2017.

[94] Sherri Davidoff. Cleartext passwords in linux memory. 2008.

[95] defintel. Shadow Puppets Domain Shadowing 101. https://defintel.com/blog/index.php/2016/03/shadow-puppets-domain-shadowing-101.html, 2016.

[96] Joe Devietti, Colin Blundell, Milo M. K. Martin, and Steve Zdancewic. Hardbound: Architectural support for spatial safety of the c programming language. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[97] Dinakar Dhurjati and Vikram Adve. Efficiently detecting all dangling pointer uses in production servers. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, 2006.

[98] Dynamic DNS. https://doc.pfsense.org/index.php/Dynamic_DNS. 2017.

[99] Forward DNS. https://scans.io/study/sonar.fdns_v2. 2017.

[100] DNSDB. https://www.farsightsecurity.com/solutions/dnsdb/. 2017.

[101] customer passwords reset Domain registrar attacked. http://www.theregister.co.uk/2013/05/09/name_dot_com_data_leak/. 2013.

[102] Kun Du, Hao Yang, Zhou Li, Haixin Duan, and Kehuan Zhang. The ever-changing labyrinth: A large-scale analysis of wildcard dns powered blackhat seo. In *USENIX Security Symposium (USENIX Security)*, 2016.

[103] David Dunkel. Catch Me If You Can: How APT Actors Are Moving Through Your Environment Unnoticed. http://blog.trendmicro.com/catch-me-if-you-can-how-apt-actors-are-moving-through-your-environment-unnoti 2015.

[104] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. ZMap: Fast internet-wide scanning and its security applications. In *USENIX Security*, 2013.

[105] Manuel Egele, Peter Wurzinger, Christopher Kruegel, and Engin Kirda. Defending browsers against drive-by downloads: Mitigating heap-spraying code injection attacks. In *Proceedings of International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2009.

[106] Mark Felegyhazi, Christian Kreibich, and Vern Paxson. On the potential of proactive domain blacklisting. In *USENIX LEET*, 2010.

[107] Security Alert: Angler EK Accounts for Over 80% of Drive-by Attacks in the Past Month. https://heimdalsecurity.com/blog/angler-exploit-kit-over-80-of-drive-by-attacks/. 2016.

[108] S. Forrest, S.A. Hofmeyr, A. Somayaji, and T.A. Longstaff. A sense of self for unix processes. In *Proceedings of IEEE Symposium on Security and Privacy*, 1996.

[109] Chris Grier, Lucas Ballard, Juan Caballero, Neha Chachra, Christian J. Dietrich, Kirill Levchenko, Panayiotis Mavrommatis, Damon McCoy, Antonio Nappa, Andreas Pitsillidis, Niels Provos, M. Zubair Rafique, Moheeb Abu Rajab, Christian Rossow, Kurt Thomas, Vern Paxson, Stefan Savage, and Geoffrey M. Voelker. Manufacturing compromise: The emergence of exploit-as-a-service. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, 2012.

[110] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in cyclone. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, 2002.

[111] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold boot attacks on encryption keys. In *Proceedings of Usenix Security*, 2008.

[112] Shuai Hao, Haining Wang, Angelos Stavrou, and Evgenia Smirni. On the DNS deployment of modern web services. In *IEEE ICNP*, 2015.

[113] Shuang Hao, Alex Kantchelian, Brad Miller, Vern Paxson, and Nick Feamster. Predator: Proactive recognition and elimination of domain abuse at time-of-registration. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[114] Shuang Hao, Matthew Thomas, Vern Paxson, Nick Feamster, Christian Kreibich, Chris Grier, and Scott Hollenbeck. Understanding the domain registration behavior of spammers. In *ACM IMC*, 2013.

[115] A. Herzberg and H. Shulman. Fragmentation considered poisonous, or: One-domain-to-rule-them-all.org. In *IEEE CNS*, 2013.

[116] Amir Herzberg and Haya Shulman. Security of patched DNS. In *ESORICS*, 2012.

[117] Amir Herzberg and Haya Shulman. Socket overloading for fun and cache-poisoning. In *ACSAC*, 2013.

[118] Tobias Holgers, David E. Watson, and Steven D. Gribble. Cutting through the confusion: A measurement study of homograph attacks. In *USENIX ATC*, 2006.

[119] Thorsten Holz, Christian Gorecki, Konrad Rieck, and Felix C. Freiling. Measuring and detecting fast-flux service networks. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2008.

[120] Threat Spotlight: Angler Lurking in the Domain Shadows. http://blogs.cisco.com/security/talos/angler-domain-shadowing. 2015.

[121] Luca Invernizzi, Stefano Benvenuti, Marco Cova, Paolo Milani Comparetti, Christopher Kruegel, and Giovanni Vigna. Evilseed: A guided approach to finding malicious web pages. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2012.

[122] Gregoire Jacob, Ralf Hund, Christopher Kruegel, and Thorsten Holz. Jackstraws: Picking command and control connections from bot traffic. In *Proc. 20th USENIX Security Symposium*, 2011.

[123] S. Jana and V. Shmatikov. Abusing file processing in malware detectors for fun and profit. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[124] Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. Shadowreplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[125] Jian Jiang, Jinjin Liang, Kang Li, Jun Li, Haixin Duan, and Jianping Wu. Ghost domain name: Revoked yet still resolvable. In *NDSS*, 2012.

[126] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the General Track of the Annual Conference on USENIX Annual Technical Conference (ATC)*, 2002.

[127] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman & Hall/CRC, 1st edition, 2011.

[128] D. Kaminsky. It's the end of the cache as we know it. In *Blackhat Briefings*, 2008.

[129] Kankanews. Xinnet breach leads false resolution of registered sites. http://www.kankanews.com/a/2014-04-02/0014513245.shtml, 2014.

[130] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*, 2015.

[131] M. T. Khan, X. Huo, Z. Li, and C. Kanich. Every second counts: Quantifying the negative externalities of cybercrime via typosquatting. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[132] Mazen Kharbutli, Xiaowei Jiang, Yan Solihin, Guru Venkataramani, and Milos Prvulovic. Comprehensively and efficiently protecting the heap. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[133] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou, and XiaoFeng Wang. Effective and efficient malware detection at the end host. In *Proceedings of USENIX Security Symposium*, 2009.

[134] Maciej Korczynski, Samaneh Tajalizadehkhoob, Arman Noroozian, Maarten Wullink, Cristian Hesselman, and Michel van Eeten. Reputation metrics design to improve intermediary incentives for security of tlds. In *Proceedings of 2nd IEEE European Symposium on Security and Privacy (Euro S&P*.

[135] Marc Kührer, Thomas Hupperich, Jonas Bushart, Christian Rossow, and Thorsten Holz. Going wild: Large-scale classification of open DNS resolvers. In *ACM IMC*, 2015.

[136] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-pointer integrity. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2014.

[137] Pavel Laskov and Nedim Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2011.

[138] How lead fraud happens? https://www.databowl.com/blog/posts/2015/10/07/how-lead-fraud-happens.html. 2015.

[139] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2015.

[140] Nektarios Leontiadis, Tyler Moore, and Nicolas Christin. Measuring and analyzing search-redirection attacks in the illicit online prescription drug trade. In *Proceedings of USENIX Conference on Security*, 2011.

[141] Charles Lever, Robert Walls, Yacin Nadji, David Dagon, Patrick McDaniel, and Manos Antonakakis. Domain-z: 28 registrations later – measuring the exploitation of residual trust in domains. In *IEEE S&P*, 2016.

[142] Chaz Lever, Platon Kotzias, Davide Balzarotti, Juan Caballero, and Manos Antonakakisz. A lustrum of malware network communication: Evolution and insights. In *38th IEEE Symposium on Security and Privacy (S&P)*, 2017.

[143] Frank Li, Zakir Durumeric, Jakub Czyz, Mohammad Karami, Michael Bailey, Damon McCoy, Stefan Savage, and Vern Paxson. You've got vulnerability: Exploring effective vulnerability notifications. In *USENIX Security*, 2016.

[144] Wei-Jen Li, Salvatore Stolfo, Angelos Stavrou, Elli Androulaki, and Angelos D. Keromytis. A study of malcode-bearing documents. In *Proceedings of International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2007.

[145] Z. Li, S. Alrwais, X. Wang, and E. Alowaisheq. Hunting the red fox online: Understanding and detection of mass redirect-script injections. In *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[146] Z. Li, S. Alrwais, Y. Xie, F. Yu, and X. Wang. Finding the linchpins of the dark web: a study on topologically dedicated hosts on malicious web infrastructures. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[147] Zhou Li, Kehuan Zhang, Yinglian Xie, Fang Yu, and XiaoFeng Wang. Knowing your enemy: Understanding and detecting malicious web advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, 2012.

[148] Ben Liblit, Alex Aiken, Alice X. Zheng, and Michael I. Jordan. Bug isolation via remote program sampling. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, 2003.

[149] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[150] Zhenhua Liu. Breeding sandworms: How to fuzz your way out of adobe reader x's sandbox. In *Blackhat*, 2012.

[151] Vitaliy B. Lvin, Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Archipelago: Trading address space for reliability and security. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[152] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond blacklists: Learning to detect malicious web sites from suspicious urls. In *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, 2009.

[153] Justin Ma, Lawrence K. Saul, Stefan Savage, and Geoffrey M. Voelker. Identifying suspicious urls: An application of large-scale online learning. In *ICML*, 2009.

[154] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In *Proceedings of ACM SIGSAC symposium on Information, computer and communications security (AsiaCCS)*, 2013.

[155] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A pattern recognition system for malicious pdf files detection. In *Proceedings of International conference on Machine Learning and Data Mining in Pattern Recognition (MLDM)*, 2012.

[156] Lets Encrypt Now Being Abused By Malvertisers. http://blog.trendmicro.com/trendlabs-security-intelligence/lets-encrypt-now-being-abused-by-malvertisers. 2016.

[157] Malware-Traffic-Analysis. 2017-04-06 - EITEST RIG EK FROM 109.234.36.165 SENDS MATRIX RANSOMWARE VARIANT. http://www.malware-traffic-analysis.net/2017/04/06/index2.html, 2017.

[158] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English shellcode. In *Proceedings of ACM conference on Computer and communications security (CCS)*, 2009.

[159] Mcafee. Net losses: Estimating the global cost of cybercrime. https://www.mcafee.com/us/resources/reports/rp-economic-impact-cybercrime2.pdf, 2014.

[160] Alexa Top 1 Million. http://s3.amazonaws.com/alexa-static/top-1m.csv.zip. 2017.

[161] Tyler Moore and Richard Clayton. The ghosts of banking past: Empirical analysis of closed bank websites. In *FC*, 2014.

[162] Mozilla. Public suffix list. https://publicsuffix.org/list/public_suffix_list.dat, 2017.

[163] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[164] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2014.

[165] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[166] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management (ISMM)*, 2010.

[167] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[168] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[169] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[170] Nick Nikiforakis, Luca Invernizzi, Alexandros Kapravelos, Steven Van Acker, Wouter Joosen, Christopher Kruegel, Frank Piessens, and Giovanni Vigna. You are what you include: Large-scale evaluation of remote javascript inclusions. In *ACM CCS*, 2012.

[171] Gene Novark and Emery D. Berger. Dieharder: Securing the heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, 2010.

[172] Gene Novark, Emery D. Berger, and Benjamin G. Zorn. Exterminator: Automatically correcting memory errors with high probability. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[173] Alexandros Ntoulas, Marc Najork, Mark Manasse, and Dennis Fetterly. Detecting spam web pages through content analysis. In *Proceedings of the 15th International Conference on World Wide Web (WWW)*, 2006.

[174] Yutaka Oiwa. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2009.

[175] Vasileios Pappas, Zhiguo Xu, Songwu Lu, Daniel Massey, Andreas Terzis, and Lixia Zhang. Impact of configuration errors on DNS robustness. In *ACM SIGCOMM*, 2004.

[176] PassiveDNS. http://netlab.360.com/. 2017.

[177] R. Perdisci, M. Antonakakis, Xiapu Luo, and Wenke Lee. Wsec DNS: Protecting recursive DNS resolvers from poisoning attacks. In *IEEE/IFIP DSN*, 2009.

[178] 116 DOMAIN CREDENTIALS STOLEN PERU DOMAIN REGISTRAR HACKED & 207. https://www.alertlogic.com/blog/peru-domain-registrar-hacked-and-207,116-domain-credentials-stolen-anonymous-group/. 2012.

[179] Daniel Plohmann, Khaled Yakdan, Michael Klatt, Johannes Bader, and Elmar Gerhards-Padilla. A comprehensive measurement study of domain generating malware. In *25th USENIX Security Symposium*, 2016.

[180] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An empirical study of real-world polymorphic code injection attacks. In *LEET*, 2009.

[181] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[182] Manish Prasad and Tzi cker Chiueh. A binary rewriting defense against stack based overflow attacks. In *In Proceedings of the USENIX Annual Technical Conference (ATC)*, 2003.

[183] Venugopalan Ramasubramanian and Emin Gün Sirer. Perils of transitive trust in the domain name system. In *ACM IMC*, 2005.

[184] CDN IP ranges. https://zenodo.org/record/842988#.wzjtrvggmzm. 2017.

[185] Avri Schneider. Whos looking for eggs in your pdf? http://labs.m86security.com/2010/11/whos-looking-for-eggs-in-your-pdf/, 2010.

[186] scikit learn. http://scikit-learn.org/. 2017.

[187] Karthik Selvaraj and Nino Fred Gutierrez. The rise of pdf malware. Technical report, Symantec, 2010.

[188] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference (ATC)*, 2012.

[189] The shadow knows: Malvertising campaigns use domain shadowing to pull in Angler EK. https://www.proofpoint.com/us/threat-insight/post/The-Shadow-Knows/. 2015.

[190] M. Zubair Shafiq, Syed Ali Khayam, and Muddassar Farooq. Embedded malware detection using markov n-grams. In *Proceedings of International conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2008.

[191] Suphannee Sivakorn, Iasonas Polakis, and Angelos Keromytis. The cracked cookie jar: HTTP cookie hijacking and the exposure of private information. In *IEEE S&P*, 2016.

[192] skape. *Safely Searching Process Virtual Address Space*, 2004.

[193] Malvertising slowing down but not out. https://blog.malwarebytes.com/cybercrime/exploits/2016/07/malvertising-slowing-down-but-not-out/. 2016.

[194] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of Annual Computer Security Applications Conference (ACSAC)*, 2012.

[195] Kevin Z. Snow and Fabian Monrose. Automatic hooking for forensic analysis of document-based code injection attacks: Techniques and empirical analyses. In *Proceedings of the European Workshop on System Security (EuroSec)*, 2012.

[196] Alexander Sotirov. Heap feng shui in javascript. In *Blackhat Europe*, 2007.

[197] Threat spotlight: CISCO TALOS thwarts access to massive international exploit kit generating $60M annually from ransomware alone. http://www.talosintelligence.com/angler-exposed/. 2015.

[198] Tom Spring. Inside the RIG exploit kit. https://threatpost.com/inside-the-rig-exploit-kit/121805/, 2016.

[199] Nedim Srndic and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *NDSS*, 2013.

[200] The story around the Linode hack. https://news.ycombinator.com/item?id=5667027. 2013.

[201] Gianluca Stringhini, Christopher Kruegel, and Giovanni Vigna. Shady paths: Leveraging surfing crowds to detect malicious web pages. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, 2013.

[202] Janos Szurdi, Balazs Kocso, Gabor Cseh, Jonathan Spring, Mark Felegyhazi, and Chris Kanich. The long "taile" of typosquatting domain names. In *USENIX Security Symposium (USENIX Security)*, 2014.

[203] Florian Tegeler, Xiaoming Fu, Giovanni Vigna, and Christopher Kruegel. BotFinder: Finding bots in network traffic without deep packet inspection. In *Proc. 8th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '12, 2012.

[204] Kurt Thomas, Chris Grier, Justin Ma, Vern Paxson, and Dawn Song. Design and evaluation of a real-time url spam filtering service. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, 2011.

[205] Talos ShadowGate Take Down: Global Malvertising Campaign Thwarted. http://blog.talosintelligence.com/2016/09/shadowgate-takedown.html. 2016.

[206] Donghai Tian, Qiang Zeng, Dinghao Wu, Peng Liu, and Changzhen Hu. Kruiser: Semi-synchronized non-blocking concurrent kernel heap buffer overflow monitoring. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2015.

[207] Hover Resets User Passwords Due to Possible Breach. http://www.securityweek.com/hover-resets-user-passwords-due-possible-breach/. 2015.

[208] Angler Attempts to Slip the Hook. http://blog.talosintelligence.com/2016/03/angler-slips-hook.html. 2016.

[209] Joseph Tucek, Shan Lu, Chengdu Huang, Spiros Xanthos, and Yuanyuan Zhou. Triage: Diagnosing production run failures at the user's site. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[210] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of European Workshop on System Security (EUROSEC)*, 2011.

[211] A Look Into Malvertising Attacks Targeting The UK. https://blog.malwarebytes.com/threat-analysis/2016/03/a-look-into-malvertising-attacks-targeting-the-uk/. 2016.

[212] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys)*, 2017.

[213] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. Memtracker: Efficient and programmable support for memory access monitoring and debugging. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture (HPCA)*, 2007.

[214] VirusTotal. https://www.virustotal.com/. 2017.

[215] T. Vissers, W. Joosen, and N. Nikiforakis. Parking sensors: Analyzing and detecting parked domains. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2015.

[216] Paul Vixie. DNS and BIND security issues. In *USENIX Security*, 1995.

[217] Peter Vreugdenhil. Adobe sandbox when the broker is broken. In *Cansecwest*, 2013.

[218] David Y. Wang, Stefan Savage, and Geoffrey M. Voelker. Cloak and dagger: Dynamics of web search cloaking. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[219] Colin Whittaker, Brian Ryner, and Marria Nazif. Large-scale automatic classification of phishing pages. In *Proceedings of the Annual Network and Distributed System Security Symposium (NDSS)*, 2010.

[220] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 2007.

[221] Wei Xu, Daniel C. DuVarney, and R. Sekar. An efficient and backwards-compatible transformation to ensure memory safety of c programs. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[222] Wei Xu, Fangfang Zhang, and Sencun Zhu. Jstill: mostly static detection of obfuscated malicious javascript code. In *Proceedings of ACM conference on Data and application security and privacy (CODASPY)*, 2013.

[223] Sandeep Yadav, Ashwath Kumar Krishna Reddy, A.L. Narasimha Reddy, and Supranamaya Ranjan. Detecting algorithmically generated malicious domain names. In *ACM IMC*, 2010.

[224] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *Proceedings of the 9th European Software Engineering Conference Held Jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, 2003.

[225] Yves Younan. Freesentry: Protecting against use-after-free vulnerabilities due to dangling pointers. In *Proc. Network and Distributed System Security Symposium (NDSS)*, 2015.

[226] Yingdi Yu, Duane Wessels, Matt Larson, and Lixia Zhang. Authority server selection of DNS caching resolvers. In *ACM SIGCOMM CCR, 42(2): 80-86*, 2012.

[227] Chuan Yue and Haining Wang. Characterizing insecure javascript practices on the web. In *Proceedings of the 18th International Conference on World Wide Web (WWW'09)*, 2009.

[228] Qiang Zeng, Dinghao Wu, and Peng Liu. Cruiser: Concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[229] Xiaofeng Zheng, Jian Jiang, Jinjin Liang, Haixin Duan, Shuo Chen, Tao Wan, and Nicholas Weaver. Cookies lack integrity: Real-world implications. In *USENIX Security*, 2015.