BACKUP TO THE RESCUE: AUTOMATED FORENSIC TECHNIQUES FOR ADVANCED WEBSITE-TARGETING CYBER ATTACKS

A Dissertation
Presented to
The Academic Faculty

By

Ranjita Pai Sridhar

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering
Department of Electrical and Computer Engineering

Georgia Institute of Technology

August 2022

BACKUP TO THE RESCUE: AUTOMATED FORENSIC TECHNIQUES FOR ADVANCED WEBSITE-TARGETING CYBER ATTACKS

Thesis committee:

Dr. Brendan Saltaformaggio, Advisor
School of Cybersecurity and Privacy and School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Frank Li
School of Cybersecurity and Privacy and School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Wenke Lee
School of Cybersecurity and Privacy and School of Computer Science
Georgia Institute of Technology

Dr. Daniel Genkin
School of Cybersecurity and Privacy
Georgia Institute of Technology

Dr. Mariusz Jakubowksi
Principal Researcher
Microsoft Research

Date approved: July 18, 2022

For my parents, Ramya and Rabindranath Pai Kasturi

And for my dearest husband, Dr. Suhasaurus Rex

ACKNOWLEDGMENTS

I feel fortunate to call myself the first Ph.D. student of Dr. Brendan Saltaformaggio and I am thankful for his mentorship. He guided me through the ups and downs of the stressful, yet a self-rewarding Ph.D. journey. I learned the art of telling the research story by paying attention to detail. I will reserve my deepest gratitude for his guidance and unconditional support toward my success. He is the best advisor that I could have asked for, and for that, I am truly grateful.

I am extremely grateful to our collaborator, CodeGuard, for sharing the dataset with us at Georgia Tech. This research would not be possible without their support.

I am deeply indebted to my seniors, Omar Alrawi and Ruian Duan, who guided me during the start of this journey. Words cannot express my gratitude to Mingxuan Yao and Jonathan Fuller, my academic siblings and my best friends. This Ph.D. has been an exercise in sustained suffering, and your company made it easier to endure. I am thankful for your friendship and I will cherish the times we spent together with our families and fur babies.

I am thankful for the unconditional support from my family. My gratitude for the endless emotional support and encouraging words from my parents, Ramya and Rabindranath Pai Kasturi, cannot be expressed in words alone. I am thankful for the love, support, and encouragement from my second set of parents from marriage, Vani and B.V. Sridhar. I am extremely grateful to Vaishali and Dr. Sudhir Pai Kasturi for the love, laughter, and prayers; for being my home away from home.

This dissertation could not have existed without my forever cheerleader, my dearest husband, Dr. Suhas Sridhar. Thank you for believing in me when I lost hope, for encouraging me to be a better version of myself every day, and for the endless supply of animal photos to cheer me on. I am proud of you for persevering and leading by example. I will always love you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Although CMSs have enjoyed rapid adoption in the industry, the security of these systems has gone severely under-considered. Despite the significant deployment of these complex software systems, to date, little research has been done to investigate and remediate CMS-targeting cyber attacks. For this reason, the vast majority of the website hosting industry has shifted to a *"backup and restore"* model of security, which relies on error-prone anti-virus (AV) scanners to prompt users to roll back to a pre-infection snapshot. This research revealed that this model is ineffective and found that the evolution of tens of thousands of attacks exhibited clear long-lived multi-stage attack patterns.

In order to make a practical impact in this space, I propose that forensic techniques must focus on the only artifact widely available to CMS owners: *the nightly backups.* To this end, this research presents TARDIS, a novel *provenance inference technique* which enables the investigation of multi-stage CMS-targeting attacks. Based on only the nightly backups, TARDIS reconstructs a timeline of the attack phases and recovers the *compromise window*, or the period of time during which the snapshots should not be trusted. I deployed TARDIS on the nightly backups of over 300K websites and found 20,591 attacks which lasted from 6 to 1,694 days, some of which were still yet to be detected.

Popular content management system (CMS) plugin marketplaces generate over a billion dollars in revenue every year [1], but little has been done by the research community to evaluate, assess, and ensure the safety of the consumers (website owners). Besides, CMS-based websites are almost entirely constructed from plugins and themes, which place implicit trust on large amounts of un-vetted code with limitless access to the webserver. This research uncovered that this trust is often broken for monetary gains and that malicious plugin authors are literally selling

plugins packed with malware to unsuspecting victims. I developed YODA, an automated framework to detect malicious plugins and track down their origin. YODA uncovered 47,337 malicious plugins on 24,931 unique websites. Among these, $41.5K had been spent on 3,685 malicious plugins sold on legitimate plugin marketplaces. Pirated plugins cheated developers out of $228K in revenues. Post-deployment attacks infected $834K worth of previously benign plugins with malware. Lastly, YODA informs our remediation efforts, as over 94% of these malicious plugins *are still active today*.

The findings from TARDIS revealed that when a webserver is compromised thousands of obfuscated and packed malware files are dropped and executed undetected alongside the webserver's existing code. Despite this surprisingly overt attack style, little research has been done to identify obfuscated/packed files on webservers, unpack them, and study their evolution over the last decade. This research studied over 10.1M obfuscated malware collected from over 27K production websites dating back to 2012. I found that 8.7M of these malware were packed and highlighted that packing has enabled the attackers to successfully evade malware detection systems. As part of this study, I developed OBIWAN, a novel dynamic analysis-based deobfuscation and unpacking methodology, which can improve server-side malware detection. In fact, OBIWAN revealed that out of 9.2M unpacked malware payloads captured from exploited webservers since 2020 *only 29% were previously unknown*, and the rest were naively repacked known (AV-detectable) malware. by obfuscated malware.

CHAPTER 1
INTRODUCTION

1.1 Dissertation Statement

The last decade has seen a significant rise in non-technical users gaining a web presence, often via the easy-to-use functionalities of Content Management Systems (CMS). In fact, over 60% of the world's websites run on CMSs. Unfortunately, this huge user population has made CMS-based websites a high-profile target for hackers. Worse still, the vast majority of the website hosting industry has shifted to a "backup and restore" model of security, which relies on error-prone AV scanners to prompt non-technical users to roll back to a pre-infection nightly snapshot. My cyber forensics research directly addresses this emergent problem by developing next-generation techniques for the investigation of advanced cyber crimes.

Driven by economic incentives, attackers abuse the trust in this economy: selling malware on legitimate marketplaces, pirating popular website plugins, and infecting websites post-deployment. Furthermore, attackers are exploiting these websites at scale by carelessly dropping thousands of obfuscated and packed malicious files on the webserver. This is counter-intuitive, since attackers are assumed to be stealthy. Despite the rise in web attacks, efficiently locating and accurately analyzing the malware dropped on compromised webservers has remained an open research challenge.

1.2 Thesis and Contributions

Before the work in this dissertation, the research community turned to fine-grained logging to understand the provenance of an attack [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Unfortunately, in the webserver ecosystem, these techniques are hardly

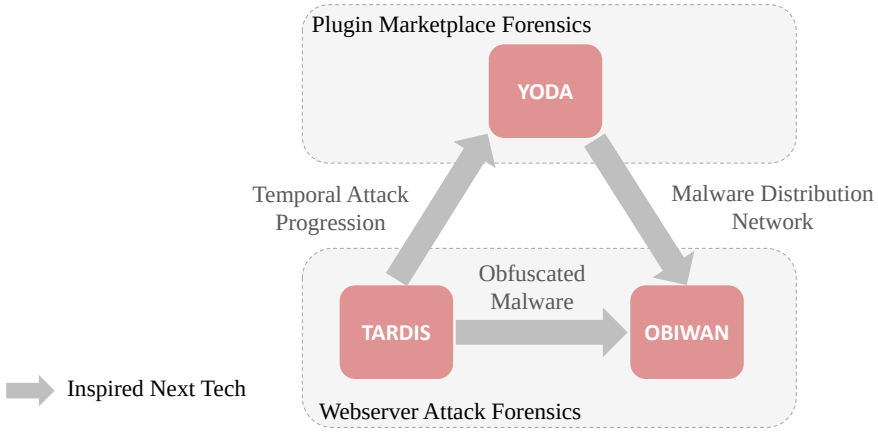

Figure 1.1: Interconnection Of The Three Components Of My Web Attack Forensics Framework.

deployed in practice due to the notable performance/space overhead incurred by these solutions [13, 14, 15, 2]. Besides, solving this problem is challenging due to the diverse range of stakeholders in the website ecosystem. Each has different motivations and visibilities into this malicious plugin problem. Website owners have full visibility over the webserver activity, but they rely on naive indicators when installing website plugins. Hosting providers have no visibility into the plugin installations but need to ensure their hosting platform remains malware-free. Plugin marketplaces have visibility over the plugins they host but need a scalable and efficient measurement of the malicious plugins being sold on their marketplaces. An ideal solution must ensure ease of use and reliable detection.

This dissertation posits that the already collected webserver nightly backup snapshots contain all required information to enable automated and scalable detection of website compromises. In this dissertation, I will present a web attack forensics framework that leverages program analysis to automatically understand the webserver's nightly backup snapshots. This will enable the recovery of temporal phases of a webserver compromise and its origin within the website supply chain. Specifically, this framework consists of three technologies that have driven this paradigm shift in web attack forensics capabilities. Figure 1.1 presents these three

components in relation to their investigation subject (webserver plugin marketplace forensics vs. webserver attack forensics) and how the technologies revealed by each influenced the development of their successors. Below I will briefly introduce these techniques, the technical contributions made by each, and the unique challenges that they overcome.

1.2.1 TARDIS

Over 60% of the world's websites run on Content Management Systems (CMS). Unfortunately, this huge user population has made CMS-based websites a high-profile target for hackers. Worse still, the vast majority of the website hosting industry has shifted to a "backup and restore" model of security, which relies on error-prone AV scanners to prompt users to roll back to a pre-infection nightly snapshot. This research had the opportunity to study these nightly backups for over 300,000 unique production websites. In doing so, I measured the attack landscape of CMS-based websites and assessed the effectiveness of the backup and restore protection scheme. To my surprise, I found that the evolution of tens of thousands of attacks exhibited clear long-lived multi-stage attack patterns. First, I will present TARDIS, an automated provenance inference technique, which enables the investigation and remediation of CMS-targeting attacks *based on only the nightly backups* already being collected by website hosting companies. With the help of our industry collaborator, I applied TARDIS to the nightly backups of those 300K websites and found 20,591 attacks which lasted from 6 to 1,694 days, some of which were still yet to be detected.

1.2.2 YODA

Modern websites owe most of their aesthetics and functionalities to Content Management Systems (CMS) plugins, which are bought and sold on widely popular marketplaces. Driven by economic incentives, attackers abuse the trust in this

economy: selling malware on legitimate marketplaces, pirating popular plugins, and infecting plugins post-deployment. This research studied the evolution of CMS plugins in over 400K production webservers dating back to 2012. I developed YODA, an automated framework to detect malicious plugins and track down their origin. YODA uncovered 47,337 malicious plugins on 24,931 unique websites. Among these, $41.5K had been spent on 3,685 malicious plugins sold on legitimate plugin marketplaces. Pirated plugins cheated developers out of $228K in revenues. Post-deployment attacks infected $834K worth of previously benign plugins with malware. Lastly, YODA informs the remediation efforts, as over 94% of these malicious plugins *are still active today*.

1.2.3 OBIWAN

Webservers remain a popular target for malware due to their outdated antivirus (AV) solutions and relatively less-technical user base. When a webserver is compromised thousands of obfuscated and packed malware files are dropped and executed undetected alongside the webserver's existing code. Despite this surprisingly overt attack style, little research has been done to identify obfuscated/packed files on webservers, unpack them, and study their evolution over the last decade. This research studied over 10.1M obfuscated malware collected from over 27K production websites dating back to 2012. My research found that 8.7M of these malware were packed and highlighted that packing has enabled the attackers to successfully evade malware detection systems. As part of this study, I developed OBIWAN, a novel dynamic analysis-based deobfuscation and unpacking methodology, which can improve server-side malware detection. In fact, OBIWAN revealed that out of 9.2M unpacked malware payloads captured from exploited webservers since 2020 *only 29% were previously unknown*, and the rest were naively repacked known (AV-detectable) malware.

1.3 Dissertation Organization

This dissertation will present the evolution of this body of work. I will highlight the progression of the shifts in web attack forensics capabilities proposed by each subsequent technology. The overall organization of this dissertation is as follows:

- Chapter 1 has introduced the the forensic benefits and unique challenges of server-side web attack investigation. I have presented an overview of my contributions to this area, with a specific focus on moving the research field away from traditional log-based provenance inference for attack evidence recovery and instead developing the concepts of spatial-temporal web attack forensics. For each component within this body of work, I have demonstrated the research problems they solve and the fundamental principles behind each technique.

- Chapter 2 explains in detail the motivation, design, implementation, and evaluation of TARDIS. I will present the landscape of web attack forensics research before the development of TARDIS, and the many investigation scenarios which benefit from TARDIS's powerful new capabilities.

- Chapter 3 introduces YODA, which focuses on identifying the origin of web attacks revealed by TARDIS. I will explore the malicious plugins that are responsible for these attacks and identify their malicious behaviors. I will explain the malware distribution network within the website plugin supply chain and the role of pirated plugins towards spreading server-side web malware.

- Chapter 4 presents OBIWAN, my most direct effort to focus on the vastly ignored analysis of obfuscated and packed server-side web malware. I will analyze the obfuscation and packing techniques adopted by attackers over the

past decade that enabled them to evade malware detection systems. I will discuss the challenges towards unpacking these malware and OBIWAN's guided unpacking technique which accomplishes this task, thus highlighting the gap in the existing defenses.

- Chapter 5 describes related research efforts which serve as motivation, background, and technical complements to my work in this dissertation.
- Chapter 6 concludes this dissertation.

CHAPTER 2

TARDIS: ROLLING BACK THE CLOCK ON CMS-TARGETING CYBER ATTACKS

Over 60% of the world's websites run on Content Management Systems (CMS) [17], with WordPress controlling nearly 60% of the CMS market [18]. Unfortunately, this widespread adoption has led to a swift increase in CMS-targeting cyber attacks. These attacks are made even easier, because CMS deployments are an amalgam of layered software and interpreters, all with varying degrees of network and system permission, which execute *on the internet-facing web server*. Worse still, this research has uncovered an unnerving trend: in-the-wild compromises of CMS deployments overwhelmingly exhibit the "low and slow" characteristics indicative of multi-stage attacks.

Despite the significant deployment of these complex software systems, to date, little research has been done to investigate and remediate CMS-targeting cyber attacks. Traditionally, the research community has turned to fine-grained logging to understand the provenance of an attack [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16]. Unfortunately, in the CMS domain, these techniques are hardly deployed in practice. Specifically, despite recent advances, fine-grained logging solutions still incur notable performance/space overhead [13, 14, 15, 2] and often require instrumenting and training with the target systems [6, 11, 3, 16]. Moreover, website owners often have no control over the underlying web server, because the entire platform is owned and maintained by a hosting provider (e.g., HostGator [19] or even a university IT department).

For these reasons, industry standard has long shifted to a "backup and restore" model of security, offered by popular platforms such as Dropmysite [20],

Codeguard [21], GoDaddy [22], Sucuri [23], and iPage [24]. Anti-virus (AV) scanners are deployed to detect compromises in websites, and nightly backups of the website's files are maintained offsite. Unfortunately, these approaches also have well-known limitations: AV signatures only catch well-known malware, they fail to detect stealthy multi-stage attacks, and high false alarm rates cause real alerts to be ignored [25, 26]. Moreover, website owners often (erroneously) revert to the most recent snapshot which did not trigger an AV alert. In fact, this research has found that website owners only take action (i.e., rollback to a snapshot) for 31% of *true alerts* and only *one-third of those* rollback to a pre-initial-infection state.

This research had the unique opportunity to study these attack trends in nightly backups from over 300,000 production websites. In collaboration with CodeGuard[1], we had initially set out to develop a website protection methodology that could replace the ineffective backup and restore standard. We began by assessing the entire history of nightly backups for 70 websites which our collaborator identified as having recently been targeted by cyber attacks. Our preliminary investigation of this dataset (detailed in §2.1) revealed something we had not expected: *the evolution of each attack exhibited clear multi-stage attack patterns* — slowly establishing an initial foothold, quietly maintaining persistence, lateral movement, cleaning up traces of earlier phases, etc.

Based on this discovery, we turned our attention to how forensic investigators could recover from these attacks. In order to make a practical impact in this space, we propose that forensic techniques must focus on the only artifact widely available to CMS owners: *the nightly backups.* To this end, this paper presents TARDIS, a novel *provenance inference technique* which enables the investigation of multi-stage CMS-targeting attacks. Based on only the nightly backups, TARDIS reconstructs a timeline of the attack phases and recovers the *compromise window*, or the period of time during which the snapshots should not be trusted.

[1]One of the largest corporate website security and backup solutions on the market.

Through our collaboration with CodeGuard, we used TARDIS to perform a systematic study of the attack landscape across 306,830 CMS-based production websites — unique domains ranging from 38 websites within the Alexa Top 10K and 4,038 in Alexa Top 1M to mom-and-pop e-commerce sites, with nightly backups covering approximately 1900 days (March 2014 to May 2019). Based on this study, we uncovered 20,591 websites (6.7%) which were compromised with advanced multi-phase attacks. Our empirical measurement revealed several concerning facts: We found that attacks persisted in CMS websites for a minimum of 6 days and a maximum of 1694 days, with a median of 40 to 100 days. More than 20% of WordPress websites, in particular, housed attacks for over a year (likely due to WordPress's significant market share). These attacks involved stealthily dropping a huge volume of malicious code affecting the web server. We found that during an attack the number of files increased by at least 50%, ranging from visitor-attacking browser exploits to full-fledged HTML-based remote control GUIs.

2.1 Preliminary Investigation

Our investigation began with 70 websites that were known to have been recently compromised. We started by asking the key cyber forensics question: How would an investigator recover the website from these attacks? Unfortunately, CMS website owners generally lack the expertise and control over the hosting server required to enable robust forensic logging. Given only these nightly backups, we quickly realized that an investigator's visibility is significantly limited.

Inferring Provenance Patterns. In trying to solve this problem, we made our first key observation: A finite number of identical provenance patterns exist within the evolution of all the websites. We first found that a file in a given snapshot can exist in 1 of 3 states: added, modified, or deleted. Figure 2.1 illustrates the three infection scenarios we observed in the website backups. A file added Ⓐ can

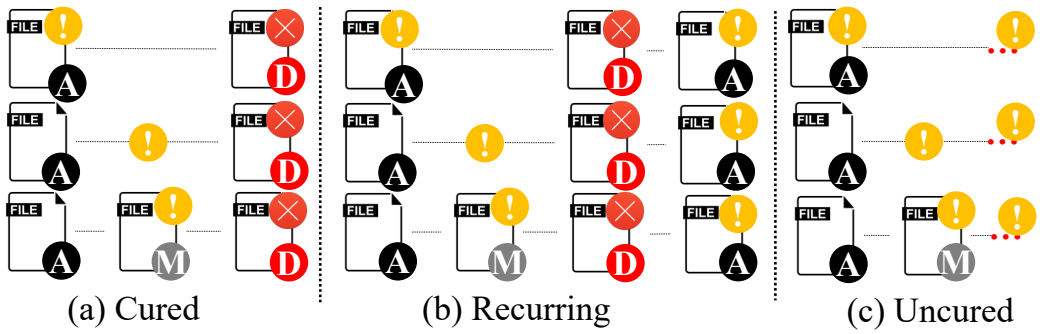

Figure 2.1: Three models of temporal infection evolution.

be flagged as suspicious (denoted by !) by an AV at some point throughout its life cycle. These files could also be flagged as suspicious (by an AV) after they are modified M. In some cases, a snapshot rollback is performed to treat the suspicious files by deleting D them. If the rollback deletes *all* of the attacker's files then the attack is cured, as shown in Figure 2.1(a). In other cases, no action is taken despite detecting a suspicious file (Uncured in Figure 2.1(c)). Unfortunately, this led to the discovery that the industry standard of "backup and restore" is entirely insufficient. We found that an alarming 80% of these websites were in fact *still infected* — many website owners had rolled back to a snapshot *and patched the vulnerability*, but given their lack of forensic expertise, they were unable to identify a pre-infection snapshot (leaving initial backdoors in place and allowing the attack to *recur*).

In order to quickly rollback to a clean snapshot, investigators must recover the *compromise window*, or the period of time during which the snapshots should not be trusted. This is further complicated by the fact that each snapshot contains tens of thousands of files (11,292 on average), making this investigation a search for needles in a haystack. Not discouraged, we drilled down into the individual snapshots from a single Drupal website, `W682886`[2], which will serve as a running example throughout this paper.

Single Snapshot Metrics. When looking into the individual snapshots from `W682886`, we made our second key observation: The complexity of each snapshot

[2]Website domain is omitted pending responsible disclosure.

Table 2.1: Temporal File Differential Analysis.

Date	Outlier	PHP	HTML	ASCII	XML	PNG	ZIP
20 Apr	-	0	0	+1	0	0	0
21 Apr	!	+7	+1	+3	+21	0	0
22 Apr	-	-2	0	+1	0	0	0
23 Apr	-	0	+2	+2	0	0	0
24 Apr	-	0	0	+1	0	0	0
25 Apr	-	+3	-1	+6	0	0	0
05 Jun	!	-13	+5	+50	0	+9	+1
07 Jun	!	-31	0	+1	0	0	0
08 Jun	!	-18	-6	-22	-20	-9	-1
09 Jun	-	+5	0	0	0	0	0
10 Jun	-	0	0	+3	0	0	0
11 Jun	-	+5	0	+1	0	0	0
12 Jun	-	+3	-7	-4	0	0	0
13 Jun	!	+9	+13	+28	+20	0	+1
14 Jun	!	-13	-13	-26	-20	0	-1
15 Jun	-	0	0	+1	0	0	0
16 Jun	-	0	0	+1	0	0	0

can be reduced to a set of measurements, called *spatial metrics*, that highlight the existence of cyber attack evidence. In addition to the state of each file in the snapshot from before (our first spatial metric), we designed another spatial metric which measures extension mismatches among the files, i.e. if a file's internal format matches the filename's extension. Similarly, we implemented another spatial metric to identify UTF-8 based code obfuscation patterns in server-side script files. For example, in the case of `W682886`, we found *3 PHP files with obfuscated payloads disguised as icon files* in the 5 June 2018 snapshot which initiated the attack. In the end, we settled on the 9 spatial metrics detailed in §4.2. These spatial metrics were effective at highlighting the presence of cyber attack artifacts within a single snapshot. However, while this was a good first step, it was neither sufficient to explain the evolution of the attack nor to understand the length of compromise.

Temporal Evolution Of Attack Phases. We collected spatial metrics to represent each snapshot of `W682886`, paying specific attention to sudden changes between pairs of consecutive snapshots. This revealed our third key observation: Modelling the *implicit events* which trigger these sudden changes can expose the attack phases. This led us to plot the temporal progression of the spatial metrics across all of `W682886`'s snapshots.

Table 2.1 shows one such progression considering only a single spatial metric, i.e. the file format numbers. The temporal evolution of this metric exposed the first attack signs. As seen in Table 2.1, sudden changes in the file format metric stand out on 21 April, 5-8 June, and 13-14 June. We found identical spatial metric outliers in 3 other Drupal websites from 14 April to 21 May 2018, suggesting the attack's lateral movement. Canali et.al. [27] also found that web attacks dropped large volumes of files on the web server, which explains the sudden changes we observed in the file format metrics. We also observed that these patterns evolved similarly over time — adding more functionality to the existing malicious code (e.g. it started with only file read capabilities, and after 8 days evolved to modify files and communicate over an SSL gateway). Eventually, we saw that these attacks tried to clean up their footprints by deleting most of the attack files.

Attack Model. These patterns formed the basis of the multi-stage attack model presented in this paper. Our study found that these attacks consisted of slow and steady attack patterns starting with establishing an initial foothold, malware injection, maintaining persistence, lateral movement, and eventually cleaning up any traces of malicious activity. This was confirmed by our case studies (§2.5), which provide an intriguing view of this widespread attack evolution.

Taken together, the above key observations drove our design of TARDIS. Modeling the temporal evolution of the spatial metrics allows TARDIS to infer the provenance of attack evidence. Further, identifying outliers within that evolution reveals both the compromise window (starting Apr 21 for `W682886`) and the progression of the attack phases. Using TARDIS, forensic investigators know where to focus their efforts and website owners can quickly revert the website to a clean snapshot. In §2.3, we will revisit these original 70 websites as manually-investigated ground truth to evaluate the effectiveness of the TARDIS framework.

Table 2.2: Formal Definitions of the State of the CMS Deployment.

Name	Symbol	Definition	Description
Time	$\Psi = (\psi, ...)$	$\Psi = (Z, +)$	Time measured in terms of the snapshot versions.
Space	$\Theta = (\theta, ...)$	$\Theta = (Z, +)$	Space of *elements* that can be monitored.
Elements	$V = (el, ...)$	$el = el\,(\theta, \psi, \psi)$	Files under investigation within their life span.
Spatial Metrics	$M = (m, ...)$	$m = m\,(\theta, \psi)$	Measurements computed against a single night's snapshot of the website backup attributes.
Labels	$L = (lb, ...)$	$lb = lb\,(\psi, \theta)$	An enumerable set of labels describing the *events* associated with the security of the *elements*.

2.2 Design

TARDIS overcomes the challenges described in §2.1 via a novel provenance inference technique, using only the nightly backups of the CMS deployment. Figure 2.2 shows the phases of TARDIS's operation: First, TARDIS constructs a temporally ordered set of spatial elements from each snapshot (§2.2.1). It then computes spatial metrics for each individual snapshot's elements (§2.2.2). This is followed by temporally correlating the collected spatial metrics and querying them against *attack models* to recover the timeline and label attack events (§2.2.3). Finally, it verifies the sequence of assigned attack labels and extracts the compromise window (§2.2.4).

2.2.1 Spatial Element Sequencing

TARDIS extracts the files associated with each night's snapshot and maps them as spatial elements ($el_j\,(\psi_i) \quad V_i$) for each snapshot $\psi_i \quad \Psi$. Here, Ψ is the set of all ψ_i, the label i denotes the index of the temporal snapshot under analysis, and j denotes the index of a spatial element in V_i. Basically, ψ_i is a point in time when the i^{th} snapshot was taken. V_i is the set of spatial elements (el_j) collected at time ψ_i. For example, the initial snapshot is collected at ψ_0, the next snapshot at ψ_1 and so on. At

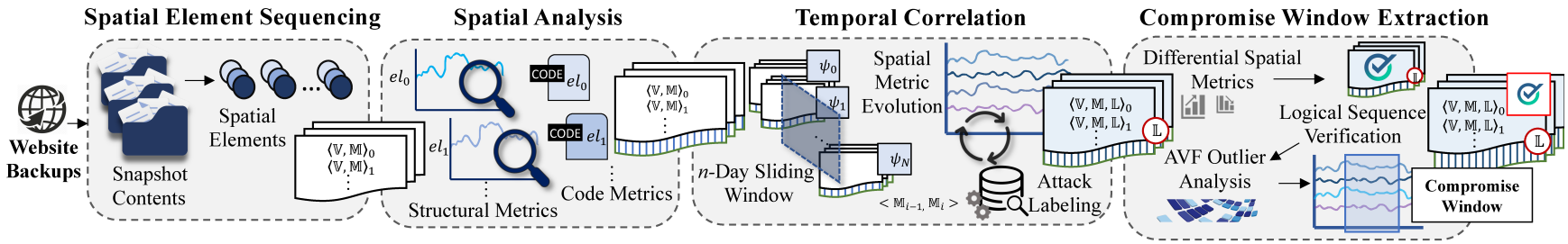

Figure 2.2: TARDIS Overview. Phase 1 constructs spatial element sets from the website backup. Phase 2 computes the structural and code metrics for each individual snapshot. Phase 3 temporally correlates the collected metrics and labels attack events. Phase 4 verifies the assigned attack labels and extracts the compromise window.

snapshot ψ_0, the set of elements are represented as $\mathsf{V}_0 = [el_0, el_1, ...]$. These elements $(el_j\,(\psi_i))$ V_i reside in the space Θ that denotes the monitoring *space* of all spatial elements (i.e., all versions of all files hosted on the web server).

While processing each temporal snapshot ψ_i, a set of initial spatial metrics $(m_k\,(\psi_i)$ $\mathsf{M}_i)$ are recorded in the set M_i. Here, the label k denotes the index of the spatial metrics collected at temporal snapshot ψ_i. These initial spatial metrics consists of the file type counts, and the state of each spatial element in terms of added, modified, or deleted. M_i is further populated with carefully selected measurements as discussed in §2.2.2. A comprehensive definition of the terminology used is presented in Table 2.2.

For example, the website `W682886`'s initial snapshot (ψ_0) contains $11,327$ files. All of these files are mapped as a sequence of spatial elements in V_0. As an example of a single spatial metric, this snapshot also contains 23 different file types (e.g. PHP, HTML, JS, CSS, etc.). This information is recorded within the spatial metric set M_0. If the backups are collected on a nightly basis for 3 months (e.g., 91 backups), then:

$$\mathsf{V} = [\mathsf{V}_0, \mathsf{V}_1, ..., \mathsf{V}_{90}]$$

$$\mathsf{M} = [\mathsf{M}_0, \mathsf{M}_1, ..., \mathsf{M}_{90}]$$

$$\mathsf{V}_0 = [el_0, el_1, ..., el_{11326}]$$

$$\mathsf{M}_0 = [num\,(PHP) = 727, num\,(CSS) = 829, ...]$$

2.2.2 Spatial Analysis

The set of spatial elements comprise of various file types (such as PHP, HTML, JS, CSS, images, plaintext, etc.), each of which requires disparate investigation techniques to identify attack attributes. To address this challenge, we split spatial analysis to extract two types of metrics: (1) structural metrics and (2) code metrics.

Structural Metrics

With the computed set of spatial elements V and the initial metrics M for each temporal snapshot, we turn to investigating this set V. Based on our observations from the preliminary study, we developed a suite of lightweight measurements that highlight the existence of suspicious elements.

Hidden Files and Directories. Long-lived multi-stage attacks can be characterized by the attacker's intent to modify the existing setup and laying low at the same time. During our preliminary study, we observed that this was achieved by dropping malicious and/or suspicious elements as a hidden file or by placing them in a hidden directory to evade first order defenses. TARDIS employs pattern matching by filtering the typically expected hidden elements (such as .htaccess) and appends a structural metric $Hide(el_j(\psi_i))$ to M_i upon finding an element $el_j \quad \mathsf{V}_i$ in a hidden location, because websites did not often employ hidden files or directories.

Extension Mismatch. We also observed that another common tactic used in CMS-targeting attacks was to disguise a server-side executable as something else. For example, we commonly observed spatial elements renamed deceptively as an icon file (e.g. favicon.ico) but containing PHP code to evade less technical CMS users. TARDIS uses the spatial element's filename to extract its extension and then matches it against the inferred file format (e.g., via the file type's magic number or other formatting that can identify the type of file). If TARDIS finds a discrepancy while matching the file type and the file extension for an element $el_j \quad \mathsf{V}_i$, it appends the structural metric $ExtMis(el_j(\psi_i))$ to M_i.

Filename Entropy. Another indicator of suspicious activity seen in CMS-targeting attacks is *long, incoherent, or randomly generated filenames*. TARDIS measures the entropy of filenames for all spatial elements el_j. A higher entropy indicates a more random filename that is less likely to be a human-generated benign filename.

Entropy is measured by password strength calculation logic [28], which computes a filename's "randomness" score by measuring its similarity to several dictionaries, spatial keyboard patterns (e.g., QWERTY, Dvorak), repetition of a single character, sequences of numbers or characters, and other commonly used keywords (e.g., `l33t`). For TARDIS, the password strength output was analogous to higher entropy (more randomness) and thus a more suspicious filename.

Since it is not possible to identify an absolute threshold for high entropy in filenames, TARDIS compares the relative entropy of the spatial elements using the median absolute deviation (MAD [29]) test. Specifically, instead of computing an absolute threshold for filename entropy, which is difficult to predict with certainty, TARDIS considers all the elements in a given temporal snapshot to first find the median entropy of all elements, followed by computing the median absolute deviations for each element and eventually checking if the median absolute deviation is greater than a relative threshold. When a relatively higher entropy is identified for an element el_j V_i from a temporal snapshot ψ_i, the structural metric $\mathit{HEntrp}(el_j(\psi_i))$ is appended to M_i.

Permission Change. TARDIS uses temporal tracking of each spatial element to detect permission changes between snapshots. In particular, when the permissions of spatial elements change from non-executable (read-only, read-write, etc.) to executable, it raises suspicion since it is unusual for a developer to start with a non-executable and provide execute privileges to it. An observation from our study was that multi-stage attacks package shell scripts in a text file and then change the permissions of the file to explore privilege escalation opportunities. Upon identifying an element el_j V_i from a temporal snapshot ψ_i with permission change equipping it with execute capabilities, TARDIS appends a structural metric $Exec(el_j(\psi_i))$ to M_i.

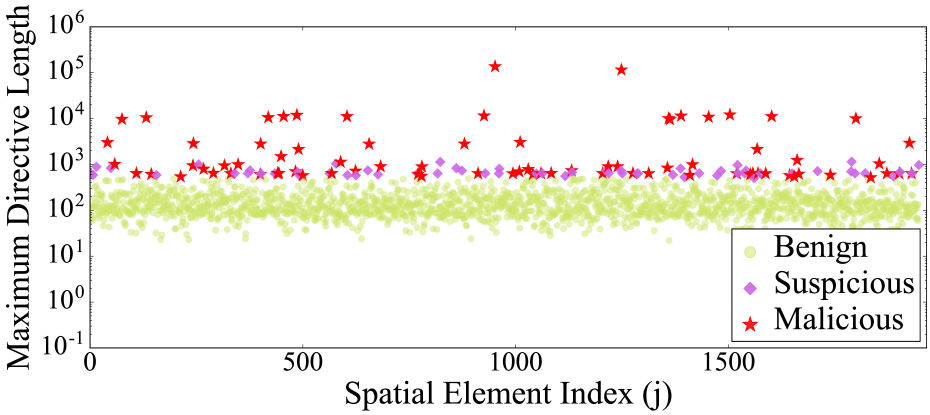

Figure 2.3: Outlier detection within the directive length distribution of all code elements in one snapshot.

Code Metrics

Since we are interested in the investigation of server-side attacks targeting CMSs, TARDIS analyzes the spatial elements containing code. These collected metrics are recorded for each snapshot ψ_i and appended to the spatial metric set M.

Script Directive Outlier Analysis. Most of the server-side source code is either part of the CMS core, associated plugins, or website-owner developed code. As they are meant to be maintained by developers, it is unusual to find source code files among the spatial elements with script directives (parsable instruction sequences) that are thousands of characters long. Hence, we observed that injecting exceptionally long and complex lines of obscure code in the spatial elements is a strong hint that can be leveraged to identify attack behaviors. Our study found that attackers use this tactic to limit the readability of injected code, delaying immediate reverse engineering attempts.

Figure 2.3 shows the directive length distribution for all spatial elements containing server-side code for `W682886`'s 2 May 2018 snapshot. The x-axis presents the spatial element index j, and the longest directive length for each of these code files is plotted along the y-axis. In benign elements (green dot) none of the directives were more than 500 characters long, whereas most attacker-injected

elements (red star) in this snapshot contained directives longer than 1500 characters. There was a mix of benign and malicious elements with maximum directive length between 500 and 1500 characters, which becomes the suspicious range (purple diamond).

Despite learning that long directives in spatial elements are suspicious, finding a threshold for directive length is not feasible due to varied coding styles and practices followed by different developers. However, it is possible to decide if a spatial element is suspicious by relatively comparing all the elements in any given temporal snapshot and performing outlier analysis. We leverage this observation to find suspicious files with relatively long directives using the median absolute deviation (MAD) previously described in §2.2.2. Upon detection of the suspiciously long directive lines in a spatial element $el_j \quad \mathsf{V}_i$ from a temporal snapshot ψ_i, TARDIS appends the code metric $LongLine(el_j(\psi_i))$ to the spatial metric set M_i.

Obfuscation Detection. We observe that server-side malware often uses a string that contains both UTF-8 characters (i.e., wide characters) and traditional 8-bit characters. While the construction of such a string itself is not malicious, it is a commonly used tactic to avoid detectors that look for known malicious string/code snippets. For example, the malicious PHP file disguised as an icon file which we mentioned earlier is included from the root of the CMS using the following *long UTF-8 (black) coupled with ASCII (red) path* to the file:

```
@include "\x2fmn\x74/s\x74or\x31-w\x632-\x64fw\x31/4\x3505\x327/\x77ww\x2
    ecv\x6dar\x61ci\x6eg.\x63om\x2fwe\x62/c\x6fnt\x65nt\x2fmo\x64ul\x65s/\
    x61gg\x72eg\x61to\x72/t\x65st\x73/f\x61vi\x63on\x5fbd\x33fd\x35.i\x63
    o";
```

Array map obfuscation is another obfuscation scheme commonly used to evade defenses [27]. An array map is defined to map each character to a different character. This map is used to deobfuscate what appears to be a jumbled list of

characters to a reverse engineer trying to make sense of this obfuscated spatial element. For example, in the following code snippet, `lnhqvwxeon()` is a function that takes a jumbled character string (in the variable `$zvkgw`) and uses the array map in `$lyfuf` to generate malicious code that gets executed as part of the PHP `eval` function:

```
$lyfuf = Array('1'=>'G', '0'=>'6', '3'=>'4', '2'=>'L', '5'=>'1', '4'=>'W',
   '7'=>'y', ...  , 'y'=>'w', 'x'=>'F', 'z'=>'l');
eval(lnhqvwxeon($zvkgw,$lyfuf));
```

Upon spatial detection of obfuscation in an element $el_j \quad \mathsf{V}_i$ from a temporal snapshot ψ_i via regex pattern matching for the cases described above, TARDIS appends a code metric $Obfus(el_j(\psi_i))$ to M_i indicating the presence of obfuscation in the element el_j.

Suspicious Payload Evaluation. In server-side spatial elements, functions such as `eval`, `base64_decode`, and `url_decode` are commonly paired to execute previously identified obfuscated code. TARDIS identifies and flags instances of the `eval` and `base64_decode`/`url_decode` pairing via pattern matching along each control flow. Upon identifying this code unwrapping technique in an element $el_j \quad \mathsf{V}_i$ from a temporal snapshot ψ_i, TARDIS appends a code metric $EvDc(el_j(\psi_i))$ to M_i indicating unsafe or suspicious code, compressed to avoid more conventional detectors.

Code Generation Capability. We observed that almost every server-side spatial element contributing to the multi-stage CMS-targeting attack contained code generation capabilities such as the use of `create_function`. Although several developers use this as part of certain CMS plugins, it is very rarely employed in ordinary server-side code development. TARDIS scouts for such code generation capabilities and appends a code metric $CodeGen(el_j(\psi_i))$ to the spatial metric set M_i upon finding an element $el_j \quad \mathsf{V}_i$ satisfying the constraints.

2.2.3 Temporal Correlation and Forensic Recovery

Based on the collected spatial metrics for each snapshot, TARDIS now attempts to temporally correlate these metrics across snapshots to identify suspicious activities that evolve within the website. Here, TARDIS is programmed to track developments over a sliding $n - day$ time window (e.g. $n = 20$ means track developments in the spatial metrics by comparing them across 20 days). In this stage, TARDIS temporally correlates the spatial metric set M_i at any temporal snapshot ψ_i with the spatial metrics M_x from all previous temporal snapshots within the sliding window ($i - n < x \leq i$) to capture the persistent adversary relationship and extract the timeline of events.

Patterns in the metrics M, assigned as a function of spatial elements, are indicative of long-lived multi-stage attack behaviors which can be detected. We construct rules to encode these behaviors based on the Boolean composition of the spatial metrics. These rules are designed to be agnostic to the individual metrics and are based on the invariants of the phases that long-lived multi-stage attacks go through. Table 2.3 shows the representative set of rules applied as part of the current implementation. Further, the temporal correlation of events encapsulating the patterns in spatial metrics is implemented by considering two consecutive temporal snapshots at a time. In particular, the 2-tuple $\mathsf{M}_{i-1}, \mathsf{M}_i$ is passed to TARDIS's temporal correlation phase (as shown in Figure 2.2) where it is queried against the attack models from Table 2.3. An attack label set L_i and a severity are assigned to each temporal snapshot, thus incrementally building the attack timeline. The assigned severity of the attack labels tells the investigator which of the labels are more critical than the others.

The rules presented in Table 2.3 capture the overall intuition behind our insights. For example, our running example `W682886` has two cases of obfuscated code injection: (1) Suspicious obfuscated code injected into an existing unobfuscated element. (2)

Table 2.3: Rules to Model Compromised CMS Events as Multi-Stage Attack Phases.

Attack Label L	**Severity**	**Attack Modeling Rule**
Establish Foothold	Medium	$ExtMis(el_j(\psi_i))\ [(el_j/\mathbb{V})\ [HEntrp(el_j(\psi_i))\ Hide(el_j(\psi_i))]]$
Obfuscated Code Injection	High	$[(size(el_j(\psi_i))>size(el_j(\psi_{i-1})))\ (MaxL(el_j(\psi_i))>MaxL(el_j(\psi_{i-1})))]\ Obfus(el_j(\psi_i))$
Malware Dropped	High	$(el_j/\mathbb{V}_{i-1})\ [Obfus(el_j(\psi_i))\ LongLine(el_j(\psi_i))\ EvDc(el_j(\psi_i))]$
Code Generation Capability	Low	$CodeGen(el_j(\psi_i))$
Defense Evasion	High	$Hide(el_j(\psi_i))\ [Obfus(el_j(\psi_i))\ EvDc(el_j(\psi_i))\ HEntrp(el_j(\psi_i))\ ExtMis(el_j(\psi_i))]$
Escalate Privileges	High	$Exec(el_j(\psi_i))\ \neg Exec(el_j(\psi_{i-1}))$
Maintain Presence	Medium	$(Sev(el_j(\psi_i))==High)\ (Sev(el_j(\psi_{i-1}))==High)$
Attack Cleanup	Medium	$(Sev(el_j(\psi_{i-1}))==High)\ [(Sev(el_j(\psi_i))==None)\ (Sev(el_j(\psi_i))==Low)\ ((el_j(\psi_i))/\mathbb{V}_i)]$

Additional obfuscated code appended to an already obfuscated element. Based on this observation, if an obfuscated spatial element $el_j\,(\psi_i)$ V_i increases in size (i.e. obfuscated attack progression), or if a script directive outlier is flagged in $el_j\,(\psi_i)$ but not $el_j\,(\psi_{i-1})$ (i.e. obfuscated code is injected into an existing unobfuscated element), and the code metric $Obfus\,(el_j\,(\psi_i))$ M_i, then an attack label "Obfuscated Code Injection" is appended to the set L_i at snapshot ψ_i. For `W682886`, we see this label assigned on 21 April, 7 June, and 13 June 2018.

Note that multiple spatial elements $el_j\,(\psi_i)$ V_i can give rise to multiple labels for each temporal snapshot. For example, there can be three spatial elements associated with $Obfus\,(el_j\,(\psi_i))$ M_i (i.e. 3 files with obfuscated code in them), and four other spatial elements associated with $ExtMis\,(el_j\,(\psi_i))$ M_i (i.e. four shell scripts disguised as gifs). In this case, both event labels *Obfuscated Code Injection* and *Privilege Escalation* are appended to the set L_i, and the highest severity of the union of this set L_i is assigned to the temporal snapshot ψ_i. It is also possible that multiple labels get assigned to a temporal snapshot due to one spatial element, i.e. an adversary can move a benign file to a hidden directory and inject it with suspicious obfuscated code. In this case, both *Defense Evasion* and *Obfuscated Code Injection* labels get appended to the set L_i, and follow the highest severity assignment as described earlier.

2.2.4 Compromise Window Recovery

With the attack labels in hand, TARDIS proceeds to extract the compromise window by parsing consecutive pairs of the 3-tuple of spatial elements, spatial metrics, and the assigned attack labels (i.e. $\mathsf{V}, \mathsf{M}, \mathsf{L}_i$). Algorithm algorithm 1 presents the pseudocode for this procedure. Lines 1-3 in Algorithm algorithm 1 describe how it takes the 3-tuple $\mathsf{V}, \mathsf{M}, \mathsf{L}$ as input, computes the differential spatial metrics $DiffAttr_i$ for each snapshot at ψ_i from consecutive pairs of $\mathsf{V}, \mathsf{M}_{i-1}$, V, M_i (e.g. recall the differential file type information shown in

Algorithm 1: Compromise Window Detection

Input: $\mathsf{V} = [\mathsf{V}_0, \mathsf{V}_1, ..., \mathsf{V}_{N-1}]$, $\mathsf{M} = [\mathsf{M}_0, \mathsf{M}_1, ..., \mathsf{M}_{N-1}]$, $\mathsf{L} = [\mathsf{L}_0, \mathsf{L}_1, ..., \mathsf{L}_{N-1}]$,
$N = Number\ of\ temporal\ snapshots$

Output: $SuspiciousRanks = [\psi_{x_0}, \psi_{x_1}, ..., \psi_{x_{N-1}}]$,
$CompromiseWindow = [\psi_{x_0}, \psi_{x_1}, ..., \psi_{x_k}]$

// Calculate frequency of each attribute value
for $\psi_i \quad \Psi$ **do**
  $DiffAttr_i \leftarrow \mathsf{V}_i - \mathsf{V}_{i-1}$, for each $el_j \quad \mathsf{V}_i$
  $DiffAttr_i \leftarrow \mathsf{M}_i - \mathsf{M}_{i-1}$, for each $m_j \quad \mathsf{M}_i$
  // Verify label sequence
  if i! = 0 *and* L_i *comes after* L_{i-1} **then**
    $CorrectLabel_i = True$
  end
end
$AttrFreq$ = Frequency of each attribute $da_j \quad DiffAttr$
// Calculate AVF scores
for $DiffAttr_i \quad DiffAttr$ **do**
  $score \leftarrow 0$;
  for $da_j \quad DiffAttr_i$ **do**
    // Score for snapshot ψ_i
    $score \leftarrow score$ + $AttrFreq[da_j]$
  end
  $AVFscores \leftarrow score/size(DiffAttr_i)$
end
$SuspiciousRanks \leftarrow$ return (sort ψ_i in order of minimum $AVFscores$)
for $\psi_i \quad \Psi$ **do**
  if $CorrectLabel_i == True$ **then**
    while $AVFScores$ *outside* $CompromiseWindow$ < $AVFScores$ *inside* $CompromiseWindow$ **do**
      $CompromiseWindow \leftarrow$ compute (range between first and last ψ_i with verified L_i)
    end
  end
end
return $SuspiciousRanks$, $CompromiseWindow$

Table 2.1).

As shown in Lines 8-16 in Algorithm algorithm 1, it then computes the attribute value frequencies (AVF) using the AVF algorithm [30] on the differential spatial metrics $DiffAttr_i$ and processes it to rank the temporal snapshots ψ_i in order of suspicious activities. The AVF algorithm performs well on categorical data with multiple attributes, the differential spatial metrics in our case [30]. In a typical AVF application, the number of anomalies to be detected are pre-programmed. Here, instead of choosing the number of anomalies to be detected, TARDIS uses the AVF algorithm to rank the temporal snapshots in the compromise window in the order of most suspicious to least suspicious.

Before TARDIS outputs the final attack labels for the entire temporal sequence, it passes the label set L through *logical sequence verification* of the associated labels (Lines 4-6 in Algorithm algorithm 1) and assesses their order of appearance. For example, when the only labels assigned are 'code generation capability' and 'attack cleanup', it has been observed that these behaviors arise from benign elements populated by the web developer and mean no harm. In such cases, the labels are retained but their severities are reduced to 'None'. If the label 'maintain presence' is seen on a snapshot prior to any other event label such as 'establish foothold' or 'malware dropped' or any other high severity modeling rule, since we know that this event sequence is intuitively not feasible, TARDIS has been programmed (again via Boolean composition of the previous label rules) to filter out sequences that do not make logical sense.

Notice that TARDIS's compromise window is only influenced by the order of 2 out of the 8 labels, i.e. *attack cleanup* and *maintain presence*. TARDIS considers all combinations of the other labels as the beginning of a compromise window. This makes TARDIS robust against attackers who might try to deploy out-of-order payloads to confound TARDIS.

Once the logical sequence of the assigned labels is verified and the temporal snapshots are ranked in the order of suspicious activities, TARDIS then identifies the *compromise window* — the period between the first and the last temporal snapshot comprising of suspicious activities with assigned and verified labels L. Also, the window is chosen such that the AVF score for every temporal snapshot outside the compromise window is higher than the score for every temporal snapshot within the compromise window (lines 17 - 21 Algorithm algorithm 1). This is the period when maximum suspicious activities take place in the website and help the investigator narrow down the analysis to a smaller window. We find that these intuitive temporally correlated spatial metrics and the attack models both align well with the design and work well in practice, as we show in §2.3 and §2.4.

For our website under investigation `W682886`, from 1 April - 30 June 2018, the compromise window is identified from 21 April - 16 June 2018. By applying the AVF algorithm, TARDIS outputs the following temporal snapshots for this website ranked in order of most suspicious to least suspicious as follows:

```
<- Most suspicious.............Least suspicious ->
5 June, 13 June, 8 June, 14 June,21 April,..,29 June
```

This aligns with our earlier visual inspection of the differential file type metrics presented in Table 2.1.

Note than these attack models are scalable irrespective of the underlying CMS, i.e. when a new tactic is identified, the TARDIS framework is designed to be highly modularized and can be easily updated to capture the essence of the new tactic and the attack label associated with it. Essentially, applying the attack modeling rules to spatial metrics and incrementally sliding along each temporal snapshot enables TARDIS to assign appropriate labels L along the compromise window, thus providing a timeline of the events as part of the long-lived multi-stage attack investigation.

2.3 Validating our Intuition

This research began with the key insight that CMS-targeting cyber attacks exhibit the "low and slow" characteristics indicative of multi-stage attacks. Based on this, we designed TARDIS to recover the compromise window and reconstruct the attack timeline. We now perform several micro-benchmarks with a ground truth set of websites to validate this intuition.

Data Set and Ground Truth. Our preliminary study in §2.1 looked at the nightly backups of 70 CMS websites which CodeGuard had identified as recently compromised. We manually investigated these websites and labeled the observed attack models. We will use these 70 websites again here as ground truth. To these 70 websites, we added the full history of nightly backups from 93 additional CMS websites, selected randomly from our collaborator's data. Again, we performed a manual investigation of these 93 new websites to obtain ground truth (discussed below). This yielded a total of 163 websites, each of which represents backups collected nightly during a 13-month period between April 2018 and May 2019.

In order to validate TARDIS's performance, we manually investigated the 163 websites to obtain ground truth. We first installed a clean version of the CMS locally and removed any file which had not been modified for each snapshot. We then searched all the code files for malware payloads and confirmed our findings with CodeGuard engineers. If our investigation found an attack, we labeled the snapshot when the attack first appeared. If the attack was cleaned up via a rollback, we labeled the corresponding snapshot. We then annotated the expected attack labels for every snapshot within that compromise window. From our 163 websites, 80 were found to be compromised. We note that this is biased by the original 70 (all of which were known-compromised) but still provides a varied test suite of benign and malicious cases.

We identified the CMS platform used by each website using WhatCMS [31] and

Table 2.4: Distribution of Compromises in the Evaluation Dataset of 163 Websites.

CMS	# of websites[1]	#GT[2]	TARDIS [3]		
			#TP[4]	#FP[5]	#FN[6]
WordPress	92	47	47	1	0
Drupal	23	15	15	1	0
Joomla	17	10	10	0	0
PivotX	9	2	2	0	0
Prestastop	2	0	0	0	0
TYPO3 CMS	8	3	3	1	0
Bourbon	4	1	1	0	0
Contao	3	0	0	0	0
Contenido	5	2	2	0	0
Total	163	80	80	3	0

1: Total number of websites evaluated for each CMS.

2: Total number of compromised websites (Groud Truth)

3: Total number of websites flagged as compromised by TARDIS.

4: True Positive, 5: False Positive, 6: False Negative.

CMS Garden [32]. The CMS market share distribution in this dataset is shown in Table 2.4. The distribution is roughly similar to the real-world distribution of CMS-based websites, e.g., a majority of websites are built on WordPress with Drupal and Joomla a close second and third.

2.3.1 Identification of Attack Models

We then drilled down into the websites identified as compromised. To validate TARDIS's attack timeline reconstruction capability, we first ran each website's sequence of backups through TARDIS and recorded what attack labels were assigned to each nightly snapshot. Note that TARDIS did not have nor need access to our ground truth for investigating the website backups, and it relied only on the temporal correlation of spatial metrics and attack models for timeline extraction. We then compared the TARDIS output attack labels with our manually derived ground truth.

Table 2.5 presents the micro-benchmark results for the 163 websites. The CMS platform is listed in Column 1. For each CMS platform, the subsequent pairs of columns show the number of websites which TARDIS marked as containing each

Table 2.5: Evaluation of the Multi-Stage Attack Phase Models.

CMS	Obf. Code Injection		Maintain Presence		Code Gen. Capability		Malware Dropped		Attack Cleanup		Establish Foothold		Defense Evasion		Escalate Privileges	
	#	#FP	#	#FP	#	#FP	#	#FP	#	#FP	#	#FP	#	#FP	#	#FP
WordPress	17	0	39	1	71	24	46	2	21	4	2	0	42	6	4	0
Drupal	6	0	15	0	18	4	13	0	9	0	0	0	12	2	5	0
Joomla	8	1	8	0	13	4	7	1	2	0	0	0	10	1	2	0
PivotX	1	0	2	0	2	0	1	0	0	0	0	0	2	0	0	0
Prestastop	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
TYPO3 CMS	0	0	2	0	3	1	3	0	1	0	0	0	2	0	0	0
Bourbon	0	0	1	0	1	1	1	0	0	0	0	0	1	0	0	0
Contao	0	0	0	0	1	1	0	0	0	0	0	0	0	0	0	0
Contenido	2	0	1	0	5	3	2	0	1	0	0	0	2	0	1	0

attack label (denoted by #) and the number of those labels which were false positive cases (denoted by #FP), i.e. our derived ground truth for that website does not contain that attack label. For example, Row 3 of Table 2.5 shows TARDIS labeled obfuscated code injection in 8 Joomla-based websites and 1 of them is an FP. Note also that any attack labels detected in known clean websites were marked as FPs.

From Table 2.5, we make several observations: Taken individually, TARDIS's attack models detect the attack labels within compromised websites with high accuracy. The labels escalate privileges and establish foothold were identified with zero FPs, as seen in Table 2.5. Also, obfuscated code injection, maintain presence, malware dropped, and attack cleanup labels saw low FP counts of 1, 1, 3, and 4, respectively, highlighting TARDIS's attack model detection accuracy.

Most importantly, when all of these attack models are considered together within a single website, TARDIS is able to prune individual false positives. TARDIS verifies the logical sequence of the recorded attack labels, computes the compromise window, and then tags the website as compromised or not, as discussed §2.2.3. This procedure pruned 38 of the 39 FPs in the code generation capability label, all 4 FPs in attack cleanup, all 3 FPs in malware dropped, 8 of the 9 FPs in defense evasion, and the only FP in obfuscated code injection, effectively removing 94.7% of the FPs listed in Table 2.5.

Another observation is that the attack tactics vary greatly across CMS platforms, but a few labels are present in *all attacks*. In particular, the maintain presence, malware dropping, and defense evasion labels are seen in all compromised CMSs. This is confirmed by our ground truth investigation. This may seem intuitive, but it confirms our premonition that *CMS-targeting attacks overwhelmingly exhibit long-lived multi-stage attack behaviors*.

Notice that TARDIS recorded 115 out of the 163 websites with code generation capabilities (a common tactic used in multi-stage attacks). However, from the Column

#FP for code generation capability label in Table 2.5, we find that in 39 of the websites, these labels were FPs. This can be attributed to the open-source nature of the CMSs and the varied coding practices followed by CMS plugin developers. The other significantly higher FP (9 out of 71 websites) is defense evasion; these are due to the presence of obfuscation used to prevent visibility into paid CMS plugins that appear like multi-stage attack behaviors at first glance.

2.3.2 Multi-Stage Attack Timeline

Based on our previous runs of TARDIS, we recorded the compromise window which was reported or "Not Compromised" for each website. We evaluate TARDIS's correctness by comparing these with our manually recorded compromise labels for the 163 websites. The results are presented in Table 2.4.

Table 2.4 shows the CMS platform and its distribution in Columns 1 and 2, respectively. Column 3 (#GT) presents the ground truth number of compromised websites in this dataset derived by manual investigation. Columns 4 through 6 show the number of websites for which TARDIS output a compromise window. Column 4 (#TP) presents the number of websites for which the TARDIS compromise window output matched the ground truth, and Column 5 (#FP) presents the number of false positives produced by TARDIS. Here, FP is essentially a "false alarm", meaning that TARDIS produced a compromise window, but the website was known to not be compromised (via our ground truth). Column 6 (#FN) presents the number of websites that are compromised and not identified by TARDIS.

Overall, TARDIS found a total of 83 websites infected with multi-stage attacks. Interestingly, more than 50% of these attacks targeted WordPress CMS, as seen in Table 2.4. In addition to the 70 known-compromised websites from §2.1, TARDIS found the 10 additional attacks in the added set of 93 websites. TARDIS reported an attack timeline that matched our ground truth for these 80 compromised websites. Manual verification confirmed the correctness of this result. To the best of

our knowledge, we did not find any websites that contained an attack that was missed by TARDIS, thus showing a zero FN count.

Notice that TARDIS produced 3 FPs, i.e. Column 4 from Table 2.4 shows 3 websites (one from WordPress, Drupal, and TYPO3 CMS). Our manual investigation revealed that all 3 websites contained user-developed security plugins with obfuscated code, similar to the tactic used by attackers, which caused TARDIS to output a compromise window for these websites. Note that there are several publicly available security plugins that contain obfuscated code (Sucuri, Wordfence, etc.), but TARDIS can handle such well-known benign obfuscation cases by checking if it belongs to a CMS security plugin with licensing information.

2.4 Deploying TARDIS in the Wild

After validating that TARDIS's analysis accurately captures the attack labels in CMS-based website backups, we worked with CodeGuard to deploy TARDIS on a significant portion of their data set. We leveraged this access to nightly backups from $306,830$ unique websites (spanning from March 2014 to May 2019) to empirically measure the health of CMS-based websites in the real world. In this section, we document our findings from using TARDIS to understand the threat landscape with respect to CMS-based websites. We are also in the process of working with CodeGaurd to inform the website owners of our findings and remediate the identified attacks.

Experimental Setup. We used a fleet of Amazon Web Services (AWS) Elastic Compute (EC2) r5.2xlarge instances with 8 virtual CPUs and 64 GB of RAM. These instances are supervised by the AWS Batch job scheduling engine to run TARDIS on hundreds of website backups in parallel.

We used several tools to assist in the investigation: Our CMS classification is built on top of WhatCMS [31] and CMS Garden [32]. TARDIS is written in Python (2500 lines of code) and leverages zxcvbn [28] for entropy estimation in the injected element

Table 2.6: Overall Distribution of Compromised Websites and Average File Counts per CMS.

CMS	Number of Websites	# Comp. Websites	Total Avg. Files Count	Only Comp. CMS	Only Benign CMS
WordPress	295,774	19,260	10,981.9	19,072.7	10,418.4
Drupal	1,340	215	17,760.0	22,288.7	16,894.5
Joomla	4,115	563	20,950.0	32,391.5	19,136.5
PivotX	509	27	28,739.7	42,075.9	27992.7
Prestashop	464	86	28,665.3	43,032.4	25,396.6
TYPO3 CMS	81	4	31,044.5	71,984.0	28,917.8
Contenido	4,543	436	16,709.8	25,851.5	15,739.3
Contao	4	0	7,634.0	NA	7,634.0

names and Pandas [33] for data analysis.

2.4.1 The CMS Landscape

Table 2.6 presents the distribution of compromises in the 306, 830 websites. Columns 1 and 2 show the CMS platform and its distribution, respectively. Column 3 shows the number of websites marked as compromised by TARDIS, i.e., the websites for which TARDIS outputs multi-stage attack labels and a compromise window. Columns 4 through 6 show the total average number of files ("spatial elements") for each CMS, the average number of files in compromised websites, and the average number of files in only the benign websites, respectively.

Table 2.6 provides interesting insights into the attack landscape of CMSs. As seen in Column 2, the majority of the websites use WordPress as their underlying CMS. In this dataset, we see that 96% of the total websites use WordPress, higher than real-world trends [18]. This is due to the high market share of WordPress users in CodeGuard's production set. From Column 2 in Table 2.6, it is evident that, except for Contao, all CMSs in this dataset are victims of multi-stage attacks. In total, we found 20, 591 compromised websites. There were 19,260 WordPress websites alone infected with these attacks (6.5% of the total WordPress websites). Interestingly, more than 16% of Joomla, 13% of Drupal, 18% of Prestashop, and 9% of Contenido websites were victims to multi-stage attacks. This goes to show that not only do

these attacks target CMSs, but they target popular and the less popular CMSs alike. In this dataset, about 5% of PivotX and TYPO CMS3 websites were compromised by long-lived multi-stage attacks, showing that these CMSs might not be popular attack targets due to their smaller market share.

As seen in Column 4 from Table 2.6, almost all CMSs contain tens of thousands of files on an average. However, an interesting metric is to compare the average number of files in compromised CMSs with those in benign CMSs. Upon comparing Columns 5 and 6, it becomes evident that invariably the attacks inject an extremely large number of files into the CMS (which we also observed during our manual investigation). As highlighted in Table 2.6, almost all the compromised websites see a 50% or more increase in files. The highest bloat in the number of files is seen for TYPO3 CMS with a 150% increase in the average number of files upon compromise. WordPress stands second, which sees an average increase of 80%.

2.4.2 Evolution of Attacks

Table 2.7 presents the distribution of attack models in the 20,591 websites that TARDIS identified as compromised. Rows 1 through 8 present these outputs for all the attack labels assigned by TARDIS. Recall that the assignment of these labels is described in Table 2.3. Columns 2 through 8 show the number of websites marked as compromised by TARDIS for each CMS. A reading from Row 4 of Table 2.7 can be interpreted as follows: After running TARDIS on a total of 295,774 WordPress websites in our dataset, it found 13,317 compromised websites with code generation capability. Lastly, Row 9 in Table 2.7 presents the total number of compromises from each CMS for comparison.

From Table 2.7, it is evident that the code generation capability is the most common tactic, seen in more 70% of all attacks, regardless of the underlying CMS. From Row 1 of this table, we see that it is not common to identify the establish foothold label in all CMSs, mainly due to the nature of our dataset. However, when

Table 2.7: Attack Phase Distribution Across the 306,830 Websites.

Phases	WordPress	Drupal	Joomla	PivotX	Prestashop	TYPO3 CMS	Contenido	Contao
Establish Foothold	339	3	34	0	1	0	22	0
Obf. Code Injection	1,629	19	29	1	0	0	39	0
Malware Dropped	7,223	141	528	11	80	0	265	0
Code Gen. Capability	13,317	188	510	27	86	4	420	0
Defense Evasion	12,491	79	63	0	0	0	181	0
Escalate Privileges	12,078	72	55	17	6	4	176	0
Maintain Presence	1,704	27	45	1	0	0	37	0
Attack Cleanup	3,795	53	123	6	6	0	84	0
Total Number of Compromised Websites	19,260	215	563	27	86	4	436	0

identified, it is a robust metric that confirms a multi-stage attack. It is also interesting to note that more than 20% of all such attacks attempt to clean up their traces after accomplishing the attack motive. However, not all multi-stage attacks actively hide their presence. As seen from Row 5 of Table 2.7, more than 60% of compromised WordPress websites try to evade defenses by following the popular hidden file/directory or the disguised file approaches. Conversely, the popular defense evasion techniques are not widely seen in compromised websites belonging to other CMSs. This could be attributed to the less-technical nature of the website owner due to which the adversaries do not spend resources on active hiding during the attack.

A significant portion of these attacks (8%) use obfuscation techniques to make it harder for the website owners to reverse engineer the injected code. Since the hosted websites cannot be taken down immediately upon detecting any traces of suspicious activity, by the time incident response can understand the obfuscated code, the adversary would have completed reconnaissance in the websites (as presented in Figure 2.4), achieved their goals, and moved towards attack cleanup.

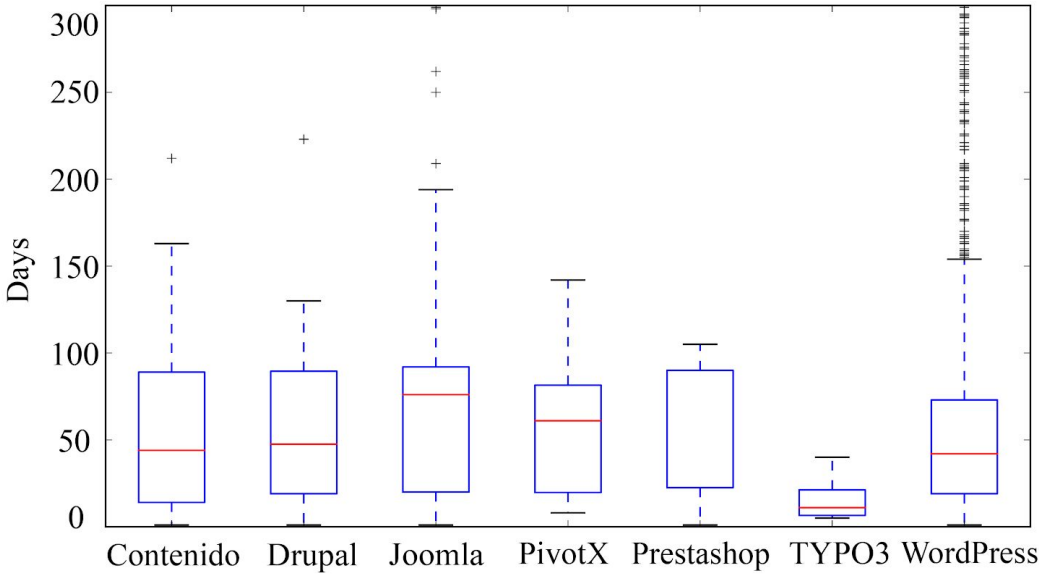

Figure 2.4: Compromise window distribution in CMSs (truncated to 300 days).

2.4.3 Compromise Window

The most important finding from this dataset was the length of compromise in CMS-based websites. Once attacked, multi-stage attacks persist in the websites for long periods of time. Figure 2.4 shows the compromise window distribution from the TARDIS output for each of the compromised CMSs. Note that this is a truncated version of the box plot to improve readability. Not shown in this figure: More than 20% of attacks on WordPress websites persist between 300 to 1694 days. As seen from Figure 2.4, most attacks in WordPress websites persist for around 40 days, as is evident from the median of the box-plot for WordPress. In comparison, the median length of the attacks is longer in Joomla and PivotX in the range of 75 to 85 days. In more than 4000 WordPress websites, these attacks persist anywhere between 3 months and 4.5 years. It is the multi-stage attacks belonging to this quartile (top 25%) that pose the most significant threat to website visitors: The dropped files simply lurk in the websites, many of which aim to exploit website visitors.

Among all CMSs, an average multi-stage attack persists the longest in Joomla, for 3 months on an average. Further, for websites that use more popular CMSs such as WordPress, Joomla, and Drupal, the attacks likely persist longer since the adversaries see a wider opportunity base and get a better return on the investment of their resources. Conversely, the less popular CMSs, TYPO3 CMS and Prestashop, are not only targeted less by persistent attacks, but those attacks also do not persist for as long. This can be attributed to the higher opportunity cost and lower returns for targeting an attack toward less popular CMSs.

2.4.4 Existing Attack Mitigation Framework

Recall, the current industry standard is a naive "backup and restore" model in conjunction with an integrated AV. Once a compromise is detected by the AV, the website owners are prompted to rollback to a previous clean snapshot. We extracted

Table 2.8: Effectiveness of the Current Industry Attack Mitigation Framework.

CMS	Infected Websites[1]	AV Alerts[2]	Rollbacks[3]	Reinfects[4]
WordPress	19,260	52	17	7
Drupal	215	9	4	4
Joomla	563	28	7	7
PivotX	27	0	0	0
Pretashop	86	0	0	0
TYPO3	4	0	0	0
Contenido	436	2	1	1
Contao	0	0	0	0
Total	20,591	91	29	19

1: # of websites compromised for each CMS, 2: # of websites with AV alerts, 3: # of websites with attempted rollbacks, 4: # of websites with reinfections after the rollbacks.

the AV reports for the dataset of 306,830 websites and instrumented TARDIS to record the number of user-initiated rollbacks and reinfections post rollback. The results are presented in Table 2.8. Note that TARDIS has no knowledge of the AV reports and relies only on its attack models for timeline reconstruction. Table 2.8 shows that AVs are ineffective in identifying almost all infected websites thus reaffirming our claim that AV signatures only catch well-known malware, and they fail to detect stealthy multi-stage attacks.

Columns 1 and 2 present the CMS platforms and the number of compromised websites from each CMS. Column 3 shows the number of infected websites from each CMS that triggered AV alerts. Column 4 presents the number of websites with AV alerts that attempted to rollback to a previous version in order to mitigate the threat identified by the AV. Column 5 presents the number of websites that attempted rollbacks and remained infected or were reinfected. As expected, the distribution of the AV alerts and the rollbacks reflect the market share of the CMSs in CodeGuard's production set, which we consider representative of CMSs at large.

Rollbacks. As presented in Table 2.8, we find it extremely concerning that among the 20,591 websites identified as compromised with long-lived multi-stage attacks by

TARDIS, *only* 91 websites see AV alerts. More so, because the website owners are alerted to rollback to a clean snapshot only when an AV alerts the website owner about a suspicious activity. This low number of AV alerts (i.e. less than 1% of the total number of compromises) is the reason why the "backup and restore" model is proving to be ineffective and these attacks persist for a significant time period.

Among the 91 websites that trigger AV alerts, not all of them take action. As seen in Table 2.8, only 29 of these 91 websites attempt a rollback to a pre-AV-alert snapshot to recover the website from the attack. Moreover, AVs are infamous for generating false alerts causing threat alert fatigue [25] — another reason why true AV alerts are ignored, perhaps explaining why only 29 of the 91 websites attempted rollbacks.

Reinfections. As seen from Table 2.8, of the 29 websites that attempted rollbacks to recover from an attack, TARDIS found reinfections in more than 65% of these websites. We imagine this was quite confusing *to the attacker* to find the website files rolled back but their original backdoors *persisted*. This confirms the long-held belief that AVs are unreliable. Not only are they missing a vast majority of the attacks, but a strong dependence on AVs is making the existing "backup and restore" technique largely ineffective. These numbers reaffirm the motivation behind TARDIS's design — the need for a systematic provenance inference technique in the space of nightly backups. We hand-verified the websites with rollbacks and found that our intuition was correct: In every case of reinfection, the rollback snapshot was inside the compromise window (identified by TARDIS) causing a reinfection. Of all the compromised websites, only 10 websites managed to rollback outside of the compromise window, thus remediating the infections. This confirms that TARDIS's provenance inference is essential for compromised website investigation.

2.4.5 Performance

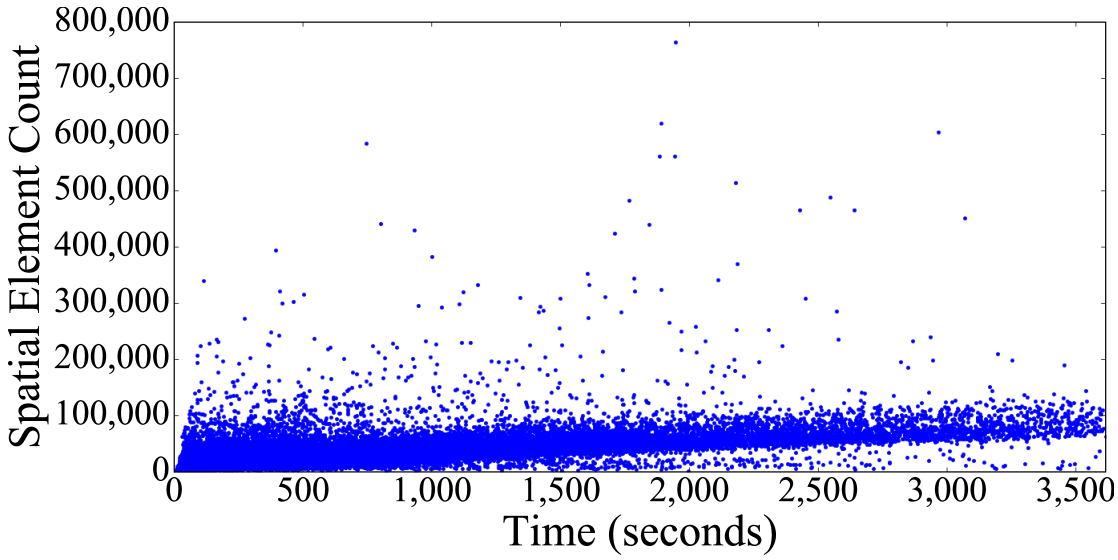

Figure 2.5: Time to process a CMS backup (seconds) versus total number of files in the CMS.

Figure 2.5 shows the time taken by TARDIS to measure all the attributes for 306, 830 websites versus the size of the websites in terms of the number of files. TARDIS linearly assesses each temporal snapshot to provide a timeline of events and event labels for the entire website with acceptable overhead. While this overhead scales with the number of files in the website (regardless of the size of these files), the increase is minimal as is seen from the gradual slope of the plot in Figure 2.5. The worst-case for TARDIS, i.e., the maximum time taken, was to process 1859 snapshots (an average of 100,000 spatial elements) is close to 3500s. As an offline forensics technique, we consider this to be quite reasonable.

2.5 Case Study

2.5.1 Case Study 1: A Global View of Attack Movement

Beyond the investigation of individual websites, deploying TARDIS *within* commercial website backup platforms [20, 21, 22, 23, 24] can provide a global view of the evolution of attack campaigns. During our experiments, we found identical

provenance evolution across 5 different WordPress websites between September and November 2018. In all of these websites, the adversary uses similar tactics of disguised *obfuscated code injection* (O) in 28 PHP files in different locations over 5 days, followed by 83 instances of *malware dropped* (M) to inject backdoor functionalities, then *maintaining presence* (P) for about 2 months, and eventually attempting *attack cleanup* (C) to remove all traces of those steps. In each of these cases, the dropped malware disables all error logging functionalities and fetches payloads from a remote server (active at the time of investigation) which it executes on the victim web server. It also collects the output buffers, sends them back to the remote server, and finally re-enables error logging. These actions were programmed to run every 48 hours.

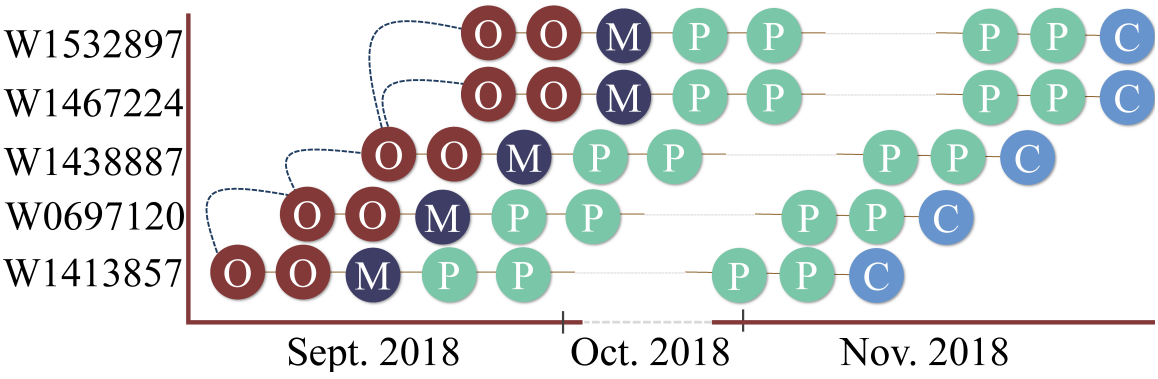

Figure 2.6: Global attack movement in WordPress websites.

Interestingly, as shown in Figure 2.6, each of these websites exhibited the same attack phase evolution and persistence for the same duration. All of these websites belong to different unique small businesses that just happen to build their website upon WordPress. Once the first WordPress website (`W1413857`) was attacked, another WordPress website (`W0697120`), completely unrelated to the initial website, exhibits the exact same injection 6 days later. This is followed by 3 other infections in three websites (`W1438887, W1467224, W1532897`) within the next 10 days. In all five of these websites, the obfuscated code injection phase lasts for 5 days, malware dropped phase for 1 day, maintain presence 51 to 56 days, and finally, the attack is cleaned up

by deleting all the injected files. Since this attack was not known to the AV, none of the attacks were flagged and the website owners did not attempt rollback.

Future Deployment. We are currently working with CodeGuard to deploy TARDIS at a global level in their backup framework to detect and track large scale attack trends. This has required expanding TARDIS to enable cross-website modeling and correlation.

2.5.2 Case Study 2: "User-Friendly" Remote Control

In a Drupal-based website, investigation of the backups for a 3 month period (Feb 2019 - Apr 2019) revealed the existence of the following phases.

```
23 Feb              Obfuscated Code Injection
24 Feb - 3 Mar      Maintain Presence
4 Mar               Malware Dropped & Defense evasion
6 Mar               Escalate Privileges
7 Mar - 12 Apr      Maintain Presence
13 Apr              Attack Cleanup
```

The integrated AV at the backup site never triggered an alert, keeping the website owner in the dark about the attack. The compromise window identified by TARDIS showed that the adversary injected obfuscated PHP code starting 23 February 2019 and maintained presence for the next few days until 3 March 2019. Starting 4 March 2019, the attacker dropped malware and used defense evasion methods: They disguised a PHP file as an icon file and uploaded a backdoor shell inside a hidden directory. On March 6, the attackers injected a full-fledged graphical user interface (GUI) for the backdoor, giving them full control of the website as shown in Figure 2.7. All of these files remained in the website for over a month. On 13 April 2019, TARDIS found that the attackers deleted the earlier injected malware to hide their previous footprints. The timeline provided by TARDIS reveals that multi-stage attack activities persisted in this website during a period of

3 months and provides a compromise window (23 February to 13 April 2019) outside of which the website can be safely rolled back. Manual investigation revealed that the CVE-2018-7600 [34] vulnerability, insufficient input sanitation on Form API (FAPI) AJAX requests, was exploited by the attacker. Note that this vulnerability was not patched until a month after this attack began.

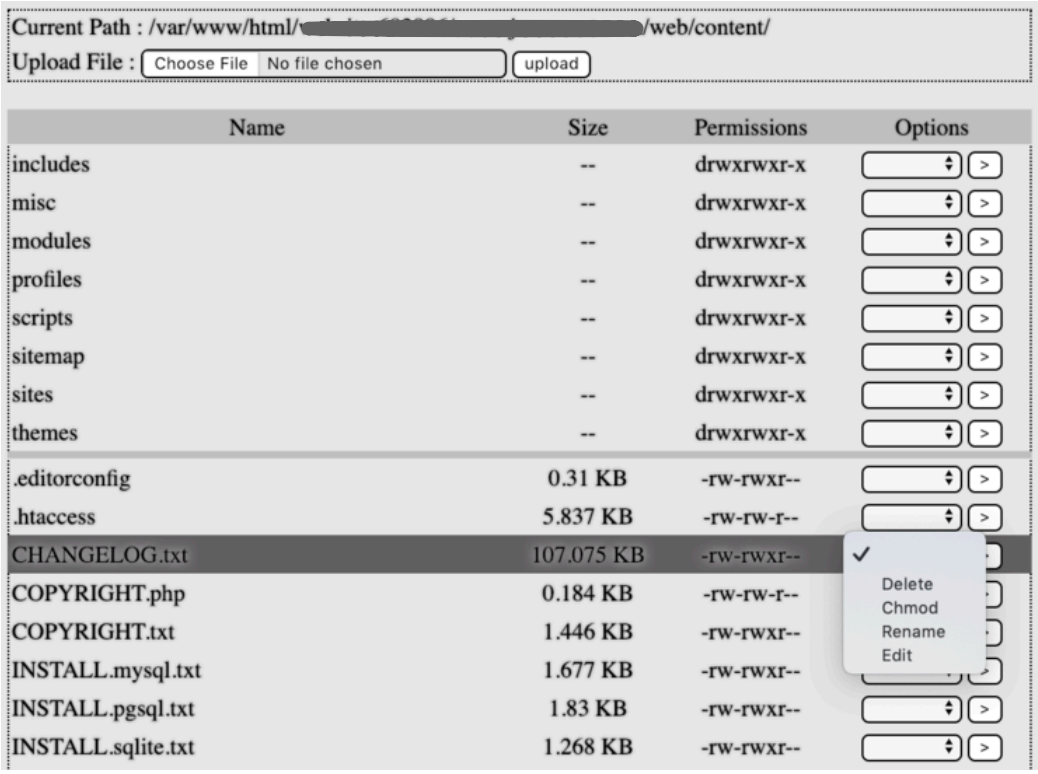

Figure 2.7: GUI backdoor injected in a Drupal website.

2.6 Limitations

The accuracy of inferring the provenance of attacks is limited by the granularity of the backups. The current industry norm is to collect website backups nightly [20, 21, 22, 23, 24]. We have shown that this is sufficient for recovering the timeline of an attack. However, if a fast-paced attack goes through multiple stages in between two consecutive backups, TARDIS would only have visibility into files at the time of the backups. Essentially, TARDIS enables website owners to calibrate between the granularity of taking backups (i.e., making TARDIS more accurate) versus requiring

a deeper manual inspection when an attack does occur.

As these multi-stage attacks evolve, TARDIS's spatial metric identification rules might need to change over time. This evolution is expected, and TARDIS's modular nature makes adding new spatial metrics straightforward. Further, TARDIS's methodology of temporal correlation of spatial metrics should stand the test of time, as it was designed agnostic to the individual metrics and is based on the invariants of the phases which multi-stage attacks go through.

CHAPTER 3

MISTRUST PLUGINS YOU MUST: A LARGE-SCALE STUDY OF MALICIOUS PLUGINS IN WORDPRESS MARKETPLACES

Many modern websites are almost entirely constructed from plugins and themes, which place implicit trust on large amounts of un-vetted code with limitless access to the webserver. Our research uncovered that this trust is often broken for monetary gains and that malicious plugin authors are literally selling plugins packed with malware to unsuspecting victims. Worse still, we found that most malicious plugins sold on popular plugin marketplaces do not employ evasion or obfuscation techniques, preferring to brazenly *hide in plain sight*.

Popular content management system (CMS) plugin marketplaces generate over a billion dollars in revenue every year [1], but little has been done by the research community to evaluate, assess, and ensure the safety of the consumers (website owners). Past research studied malicious apps in the Google Play Store [35], malicious extensions on the Chrome Web Store [36, 37], and malicious packages in package registries [38]. Prior work also exposed malicious behaviors on webservers, such as the presence of vulnerabilities [39, 40], webshells [41], and backdoors [42], but none analyzed the underlying plugins which lead to many of these attacks. Further, the complexities of prior research solutions have prevented the average CMS-user from adopting them.

CMS website owners often rely on simple indicators such as plugin popularity, ratings, and reviews on the plugin marketplaces to determine that a plugin is safe to install on their website [43]. The diligent CMS-user may consult freely available [44] or commercial [45, 46] plugin vulnerability scan databases before installing a plugin. Unfortunately, these sources provide neither complete nor robust measures of security.

Driven by economic incentives, attackers *buy the codebase* of popular free plugins, add malicious code, and wait for plugin users to auto-update [47]. In such cases, none of the commonly used simple indicators can help prevent malware from infiltrating the website.

Our research performed a global measurement of the malicious WordPress plugins ecosystem. We worked with CodeGuard[1], to analyze the WordPress plugins in over 400,000 unique webservers dating back to 2012. We uncovered 47,337 malicious plugin installs on 24,931 unique websites. Even worse, 3,685 of these plugin instances were sold on legitimate plugin marketplaces. Tracking the webservers and plugins over 8 years gave us a unique vantage point to study the temporal evolution of malicious plugins from a global perspective. We found that the number of malicious plugins on websites has steadily increased over the years, and malicious activity peaked in March 2020. Shockingly, 94% of the malicious plugins installed over those 8 years *are still active today*.

Throughout our study, we found that solving this problem is challenging due to the diverse range of stakeholders in the CMS plugin ecosystem. Each has different motivations and visibilities into this malicious plugin problem. Website owners have full visibility over the webserver activity, but they rely on naive indicators when installing plugins. Hosting providers have no visibility into the plugin installations but need to ensure their hosting platform remains malware-free. Plugin marketplaces have visibility over the plugins they host but need a scalable and efficient measurement of the malicious plugins being sold on their marketplaces. An ideal solution must ensure ease of use and reliable detection since plugins could be malicious anywhere in this supply chain: from the source marketplace to a post-deployment web attack (i.e., fake plugin injection).

To address these challenges, we developed YODA, an automated framework to

[1]One of the largest corporate website security and backup solutions on the market.

identify malicious plugins and their origin. Towards usability, this can be integrated as part of the webserver hosting platform or deployed by the plugin marketplace. Website owners are often unaware of the plugins installed *or injected* into their website, so when deployed by a hosting provider, YODA starts by detecting a webserver's (possibly hidden) plugins. YODA also crawls popular CMS plugin marketplaces to identify each plugin's provenance, ownership, and global impact. Using YODA, website owners and hosting providers can identify malicious plugins on the webserver; plugin developers and marketplaces can vet their plugins before distribution.

Our 8-year study using YODA revealed several concerning facts: While the website owners trusted the plugin ecosystem and spent a total of $7.3M on *only the plugins in our dataset*, we found that this trust is often broken for the attackers' monetary gains. Attackers impersonated benign plugin authors and spread malware by distributing pirated plugins. YODA found 1,354 instances of pirated plugins responsible for one of the largest known malvertising campaigns [48], many of which are still active today. Furthermore, $41.5K was spent on malicious plugins sold on legitimate plugin marketplaces, and plugins that cost a total of $834K were infected post-deployment by attackers. We hope that YODA can regulate and reinstate the trust between all stakeholders in the plugin ecosystem. Lastly, we have made YODA's source code available at: https://cyfi.ece.gatech.edu/

3.1 Preliminary Study: Perilous Economy

Plugins are groups of files that work together to add aesthetic features and functionalities to a CMS website. Upon each visit to the website, the CMS loads all active plugins (i.e., executes plugin code) on the webserver. **Our Dataset.** Our collaboration with CodeGuard furnished access to the nightly backups of over 400K unique WordPress websites. These backups contain the server-side files and their

	Marketplace	#Plugins		Downloads Range			Cost of Plugins			Money Spent	
		Total	Unique	Min.	Avg.	Max.	Min.	Avg.	Max.	Dataset	Global
Free Plugins	WP Plugins [49]	5,276,450	27,430	14	7.5M	260M	-	-	-	-	-
	WP Themes [50]	506,342	5,450	3	293K	7.4M	-	-	-	-	-
	Github [51][1]	448,324	5,155	0	3	9	-	-	-	-	-
Paid Plugins	CodeCanyon [52][2]	38,060	2,428	1	1.2K	9K	$2	$32	$1,104	$1.2M	$82.6M
	ThemeForest [53]	61,574	5,837	1	29.6K	611K	$2	$84	$499	$5.1M	$31.3M
	WPMU DEV [54][3]	5,984	110	55.4K	1.4M	10.5M	$15	$63	$190	$370K	$96.4M
	EDD [55][4]	13,123	245	-	-	-	$6	$49	$199	$643K	-
	Total	5,782,783	43,621	1	939K	260M	$2	$63	$1,104	$7.3M	$210.3M

1: Since Github does not provide the repository download info, we used the number of stars as a measure of popularity.

2: We found a plugin, Choco Drops [56], on CodeCanyon for $10,000,003. Since this is an outlier, it has been excluded here.

3: WPMU DEV charges a $15 monthly subscription to use any plugin on this marketplace. The price range reflects the yearly cost.

4: Downloads range was not publicly available for EDD and has been excluded in this table.

Table 3.1: The Economy of WordPress Plugin Marketplaces.

version-controlled changes collected from July 2012 to July 2020. They give YODA the vantage point of both an individual website owner as well as a hosting provider (i.e., access to the webserver files). This allows us to retroactively deploy YODA over 8 years by executing YODA on each nightly backup for every website. Note that CodeGuard anonymized the website owner profiles — only a random ID and the URL were linked to each website backup. Furthermore, we only analyzed the webserver files with no database access. All websites in our study are CodeGuard's active clients, i.e., if a client stops using CodeGuard's service all their data is immediately deleted, thus we will not have access to it.

Responsible Disclosure. The individual website-owners are anonymized by CodeGuard. However, all of CodeGuard's customers agree to their Privacy Policy whereby their data may be shared with third-parties to help CodeGuard safeguard their websites. Since we cannot directly contact the affected victims, we have alerted CodeGuard about our findings and they are processing the disclosure.

Plugin Marketplaces. WordPress plugins and themes generate millions of dollars in sales every year [1]. These plugins[2] are either created by an individual or teams of developers, including WordPress themselves. After detecting the plugins in our dataset (5.7M of them), we performed a preliminary study to understand the scale of this economy. We measured the plugin downloads and price data in July 2020 for the plugins in our dataset. This required scraping and cross-correlating data from the plugin code and online marketplaces. Table 3.1 lists the plugin marketplaces, the total number of plugins, and the unique number of plugins from these marketplaces in our dataset. As seen in Table 3.1, thousands of plugins are freely available on the WordPress repositories [49, 50] and software development platforms, such as Github [51]. In rare cases (below 0.5% in our dataset), some plugins are available on multiple marketplaces.

[2]Plugins and themes are together referred to as plugins.

Paid versions of the plugins are sold through marketplaces, e.g., ThemeForest [53], CodeCanyon [52], and Easy Digital Downloads (EDD) [55]. Here, individual plugins are sold for as little as $2, while the bestselling plugins are valued at around $63. Table 3.1 Columns 4-6 highlight the plugin popularity in terms of the number of downloads. WP Plugins [49] is the most popular marketplace overall, with 7.5M average downloads per plugin. Some marketplaces do not sell individual plugins and instead provide a subscription service for all plugins at a flat rate. For example, WPMU DEV [54] has a $49/month subscription and is the most popular paid plugin marketplace in our dataset with 1.4M average downloads per plugin. Less-popular plugins are also directly available from freelance developers or small businesses [57, 58, 59].

Since plugin marketplaces do not provide any price history, we used the reported download counts and the prices from July 2020 to estimate the money spent on these plugins. Table 3.1 shows that website owners from our dataset alone spent $7.3M at plugin marketplaces, and we estimate the revenue earned by these plugins globally to be over $210M based on a conservative estimate of the reported download counts. We found several plugins sold with an extended license for more developer support time. We considered the regular support pricing to estimate a lower-bound for spending in the plugin ecosystem. Our estimate is also confirmed by themeshunter.com, one of the biggest WordPress themes catalogue on the market [60].

Nulled Marketplaces. Since most paid plugin marketplaces do not offer a trial option, several marketplaces started a *"try before you buy"* initiative. Unfortunately, this gave rise to pirated "trial plugin" marketplaces, referred to as nulled marketplaces. Nulled plugins are pirated versions of originally paid plugins, freely distributed via nulled marketplaces (unbeknownst to the original creator). Generally, these plugins have been hacked or contain modified code to cause user harm or collect sensitive user data and made to work indefinitely without a license

key [61]. Our study has found that, more often than not, nulled plugins introduce malicious code onto webservers (§3.4.3).

Insufficient Market Oversight. While these marketplaces are growing rapidly, the regulations to assess plugins are minimal. For example, as mentioned in Table 3.1, our study found a CodeCanyon plugin for $10,000,003 [56] with a note from the plugin author to not buy the plugin. The fact that the plugin author was able to set the price so high, to dissuade downloaders, rather than legitimately removing the listing underscores how little oversight these marketplaces provide. We also found that attackers include malicious behaviors in plugins then sell them on reputable plugin marketplaces, such as the WordPress repository. A report by Wordfence [47], a leading WordPress malware scanner, found nine popular plugins updated at source (i.e., the WordPress plugin store) with malicious code as part of a coordinated spam campaign (see §3.6). It is more urgent now than ever to study the impact of this problem and address the challenges toward securing the plugin ecosystem.

3.2 Design

Figure 3.1 shows an overview of the YODA pipeline. YODA first conducts Plugin Detection (§3.2.1). Hosting providers and website owners can deploy this to detect plugins on their websites. But, marketplaces or plugin developers can skip directly to YODA's Malicious Behavior Detection (§3.2.2) for a given plugin. Next, YODA identifies the Origin of Malicious Plugins (§3.2.3). Finally, it performs an Impact Study (§3.2.4) to understand the scale and impact of the plugin economy.

3.2.1 Plugin Detection

YODA detects all of the webserver's plugins by identifying the plugin root and all the associated files that belong to the plugin. Reliably detecting plugins based on the webserver files alone can be challenging because CMSs provide limited guidelines leading to a lack of code consistency. CMS-users (plugin developers, website owners,

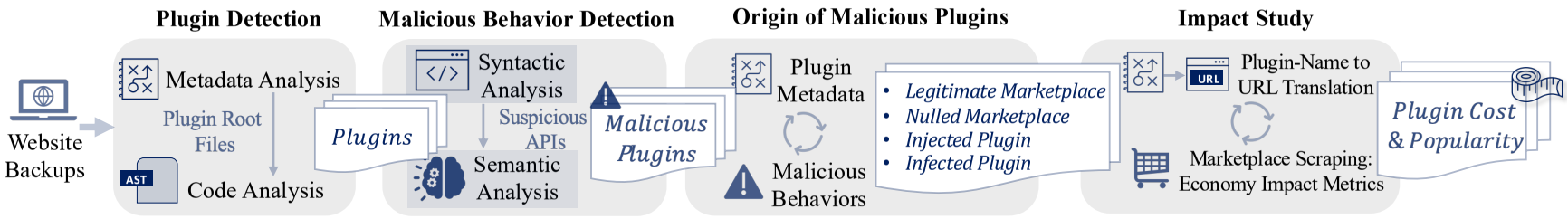

Figure 3.1: YODA Design Overview.

and attackers) often customize their plugins and do not follow coding guidelines. CMSs provide directories to maintain plugins and themes, but users often place them in random locations on the server, so we cannot solely rely on the directory structure to reliably detect plugins.

To address these challenges, YODA performs (1) *metadata analysis* to identify the plugin root files and (2) *code analysis* to identify all of the associated files as part of the plugin (shown in Figure 3.1).

Metadata Analysis. YODA parses the comments from all of the server-side code files and performs regular expression matching to identify the plugin root files (i.e., files containing the plugin header, a specially formatted block comment that contains metadata about the plugin). This ensures that bad coding practices and possibly hidden plugins injected by attackers do not go undetected.

```
1  /**
2   * Plugin Name:       My Basics Plugin
3   * Plugin URI:        https://example.com/plugins/the-basics/
4   * Description:       Handle the basics with this plugin.
5   * Version:           1.10.3
6   * Requires at least: 5.2
7   * Requires PHP:      7.2
8   * Author:            John Smith
9   * Author URI:        https://author.example.com/
10  * License:           GPL v2 or later
11  * License URI:       https://www.gnu.org/licenses/gpl-2.0.html
12  * Text Domain:       my-basics-plugin
13  * Domain Path:       /languages
14  */
```

Figure 3.2: A Typical WordPress Plugin Header.

A typical plugin header is shown in Figure 3.2. It is a specially formatted PHP/CSS block comment that contains metadata about the plugin, such as its name, author, version, license, etc. At the very least, the header comment must contain the Plugin Name. An attacker could try to evade YODA's plugin detection by dropping the plugin header. However, this would work against the attacker: without a valid plugin header containing at least the Plugin Name metadata, the plugin will never get loaded by WordPress core and hence remain dormant on the

webserver.

```
has_filter, add_filter, apply_filters,
apply_filters_ref_array, do_action_ref_array,
remove_filter, remove_all_filters,
doing_filter, has_action, add_action, do_action,
remove_action, remove_all_actions, doing_action,
register_uninstall_hook, current_filter,
register_deactivation_hook, did_action,
register_activation_hook
```

Figure 3.3: List of WordPress Plugin APIs.

WordPress plugins are installed in dedicated plugin directories on the webserver. Figure 3.3 shows the plugin-specific API calls commonly used in WordPress plugins and themes. These are called *Hooks* that enable one piece of code to interact/modify another piece of code at specific, pre-defined spots, thus helping the plugin interact with the WordPress core.

It is up to the website owner to set the plugin's user access permissions [62]. Based on these permissions, the website owner controls the directories that a plugin can access for read, write, and execute. Plugins can either be manually updated or set to auto-update upon which WordPress downloads and installs any updates from the plugin store. However, WordPress cannot automatically update plugins from 3rd-party marketplaces. These need to be manually updated. Since most plugins have both read and write permissions in the plugin installation directory, a malicious plugin can scan for other plugins to inject malicious code, thus infecting it. Malicious plugins installed outside the plugin installation directory can use a CMS account or webshell to first change the access permissions (such as in SSO Backdoor attacks) and then infect other plugins.

Note that WordPress only identifies and loads plugins with a header, and all plugins must contain a single plugin root. If an attacker tries to evade YODA's detection by dropping the header, the plugin will not be loaded by the WordPress

core and remain dormant on the webserver. For every plugin root, YODA extracts and records the plugin metadata from the header, including the plugin name, plugin URI, author name, author URI, and plugin version. As we will see later (§3.3.1), we use this plugin version to understand how many CMS-users maintain their plugins updated to the latest version.

Code Analysis. With the plugin root files identified, YODA proceeds to find all associated plugin files P_i, i.e., $\mathsf{P}_i = \{f_{i_1}, f_{i_2}, ...\}$ where f_{i_j} is the j^{th} file in plugin P_i. To do this, YODA generates and parses the abstract syntax tree (AST) of all of the server-side code files in parallel and sub-directories of the plugin root. Since several CMS-users customize their plugins either by using configuration files or explicitly modifying the PHP code, YODA will detect f_{i_j} based on three scores, listed in decreasing order of importance. We use constant weights (3, 2, and 1) coupled with inverse exponentials to rank these scores since it is a common approach for ranking program modules [63]. While we could have chosen any decreasing range, we found that this combination produced distinct ranges that identify plugin files from non-plugin files.

1. Header Score. The existence of a plugin header n_j in a file f_{i_j} (computed during metadata analysis) is used to derive the header score h_j (weight = 3). Here, n_j can take values 1 (for plugin root with a single header) or 0 (for the associated files with no header). However, the header score is included in the reliability score with the highest weight to ensure a group of plugin-like files with no header is not incorrectly identified as a plugin.

$$Header\ Score \qquad h_j = 3 \quad n_j$$

2. Reference Score. YODA uses the number of reference calls m_j linking other files as part of the plugin to derive the reference score r_j (weight = 2). Each score

is a sum of the individual contributions from all of the linked files towards the entire plugin. This contribution is scaled by an integer weight in the numerator to model the importance of linked files and an exponential in the denominator to account for a large number of referenced files. Here, the exponent x starts with n_j to further ensure the reference score contribution is lower than the header score.

$$Reference\ \ Score \qquad r_j = \sum_{x=n_j}^{n_j+m_j-1} \frac{2}{2^x}$$

3. API Score. The number of occurrences of plugin-specific API calls l_j (the full list of APIs is shown in Figure 3.3) is used to derive the API score a_j (weight = 1). Since APIs alone are insufficient to detect a plugin, l_j is scaled using a weight of 1 (numerator). Here, the exponent x starts with $n_j + m_j$ to ensure the API score contribution is lower than the header and reference scores.

$$API\ \ Score \qquad a_j = \sum_{x=n_j+m_j}^{n_j+m_j+l_j-1} \frac{1}{2^x}$$

The sum of all three scores for all plugin files in parallel and sub-directories of the plugin root is divided by the upper bound of this sum to calculate the reliability score R_i for each plugin P_i.

plugin files f_{i_j} P_i,

$$Reliability\ \ Score \qquad \mathsf{R}_i = \frac{\sum_{f_{i_j}} (h_j + r_j + a_j)}{2 \quad max(h_j, r_j, a_j)} \quad 100\%$$

The reliability score is a measure of the likelihood of a group of files being part of a plugin. If this score is greater than 95% for a group of files, YODA detects it as a plugin. We set the strictest possible threshold because we found that for a true positive plugin this score is always >98% and <55% otherwise.

An additional challenge (a special case of the above) is child plugins. They are extensions of the original plugin that enable the website owners to add customization,

Malicious Behavior	Semantic Models
Webshell	$Super\_Global[input] \rightarrow Exec$
Post Injection	$(URL \quad Blacklist) \quad (URL \rightarrow Download \rightarrow Add\_Post)$
Input Gating	$Super\_Global[password] \rightarrow Conditional \rightarrow Exec$
SSO Backdoor	$Create\_User \rightarrow Chng\_User\_Perm \rightarrow Register\_User \rightarrow Redirect\_NewUser\_Admin\_URL$
Library Function Exists	$Conditional \rightarrow Func\_Exists \rightarrow Create\_Func$
Spam Injection	$(URL \quad Blacklist) \quad (URL \rightarrow Download \rightarrow Add\_Content)$
Code Obfuscation	$(Jumbed\_Obfus \quad Long\_Line) \quad (Decode \rightarrow Exec)$
Blackhat SEO	$Conditional \rightarrow SE\_Bots \rightarrow (URL \quad Blacklist) \quad Download \rightarrow Replace\_Content$
Downloader	$(URL \quad Blacklist) \quad (URL \rightarrow Download)$
Function Reconstruction	$(Str1, Str2, .., StrN) \rightarrow Concat \rightarrow Create\_Func$
Insert User	$Create\_User \rightarrow Register\_User$
Malvertising	$(URL \quad Blacklist) \quad (URL \rightarrow Download \rightarrow (Redirect \quad Insert\_Popup))$
Fake Plugin	$Conditional \rightarrow Super\_Global[Str] \rightarrow Decode \rightarrow Exec \rightarrow Delete\_Payload$
Cryptominer	$(URL \quad Blacklist) \quad (URL \rightarrow Download \rightarrow File\_RW \rightarrow Chng\_File\_Perm \rightarrow Exec)$

Table 3.2: High-level Dataflow Sequence of the Semantic Malicious Behavior Models from Source to Sink.

i.e., modify functionalities without having them disappear after an upgrade. YODA handles child plugin detection by recursively searching through plugin sub-directories to find plugin roots and storing them as separate child plugins under their respective parent plugins.

Effective Plugin State. Since YODA can retroactively run on temporal webserver snapshots, it records the effective plugin state in each of these snapshots. For each plugin, YODA uses the individual file states of all plugin files, i.e., added (`'A'`), modified (`'M'`), no change (`'NC'`), or deleted (`'D'`) to derive the effective plugin state that could also take one of the four values: `A`/`M`/`NC`/`D`. If all individual plugin file states are added, deleted, or no change, then the effective plugin state is `'A'`, `'D'`, or `'NC'`, respectively. All other individual file state combinations produce an effective plugin state `'M'`.

3.2.2 Malicious Behavior Detection

Preliminary Study. We started by analyzing all plugins from 85 known-compromised website backups taken between April 2018 and June 2020. CodeGuard provided this subset based on signature-based AV alerts for well-known web malware. We also referenced all reports of malicious plugins being removed from popular marketplaces between 2013-2018 [64, 65, 47, 66] to identify and collect available malicious plugin samples. Since most of the removed plugins were not accessible on the marketplaces, we used the plugin name and version to scan our dataset and collect all additional malicious plugin samples[3].

We manually investigated all the plugins from above and identified 14 distinct malicious behaviors, listed in Table 3.2 Column 1. We will describe the modeling of these behaviors in the rest of this section. We also found that each of these behaviors had multiple implementations and using rule-based syntactic detection alone would quickly leave the rules obsolete. Further, state-of-the-art web malware detection relies

[3] Available at: https://cyfi.ece.gatech.edu/.

on structure-aware semantic features (e.g., code implementations of webshell features) within a single code file [67, 68]. Existing techniques do not consider the interactions between file groups. This is necessary because attackers distribute malicious behavior implementations across multiple plugin files, thus evading existing techniques.

YODA addresses these challenges by employing both syntactic features (e.g., file meta-data, sensitive APIs) and context-aware semantic features of all plugin code files (e.g., AST with resolved file dependencies).

Syntactic Analysis. Syntactic analysis uses data flow analysis to identify suspicious APIs being used as sinks in plugin code files. YODA generates the AST for all of the plugin code files and parses it to record the sensitive API classes summarized in Table 3.3. These APIs will later form the sinks for semantic dataflow analysis. Table 3.3 shows the notations used for the classes of suspicious sinks and the API class description. For example, the *Decode* class denotes decode functions such as `base64_decode, json_decode`. The list of these sinks was identified by studying past research as well as industry reports of malicious plugins being removed from popular marketplaces between 2013-2018 [64, 65, 47, 66]. YODA may miss identifying sinks that are based on novel implementations of the attack behaviors such as updates to the PHP language. Since this is a rare event, YODA's models can be updated easily when necessary.

Plugins can have inter-file dependencies that invoke suspicious APIs indirectly. An intuitive solution for handling inter-file dependencies is to analyze each plugin code file together with its dependencies, but this may lead to the repeated analysis of common dependencies and possible resource exhaustion given too many dependencies. Therefore, to increase efficiency and reduce failures YODA eliminates the inter-file dependencies by recursively replacing the dependencies with their respective ASTs via modularized API usage analysis which analyzes each dependency only once. This also handles the case of cyclic dependencies if any.

Classes of Suspicious Sinks	Suspicious API description
$Exec$	Execute code
$File\_RW$	File read/write
$Decode$	Decode functions
$Download$	Download from URLs
$Create\_Func$	Define function from string inputs
SE_Bots	References to search engine botnames
$Chng\_File\_Perm$	Changes the file permissions
$Chng\_User\_Perm$	Changes the user permissions
$Create\_User$	Creates a default user account
$Register\_User$	Registers a user account to the CMS
Add_Post	Adds a new post
$Inseert\_Popup$	Adds code to display a popup
$Add\_Content$	Appends content to HTML metadata
$Replace\_Content$	Replaces old content with new content
$Func\_Exists$	Check if a function exits
$Redirect$	Redirects to the URL passed as input
$Delete\_Payload$	Deletes the downloaded payload
$Redirect\_NewUser\_Admin\_URL$	Redirects to the new user's admin URL

Table 3.3: Classes of Suspicious API Sinks.

The dynamic nature of the PHP language (e.g., dynamically evaluating code) can introduce challenges to accurately detecting the suspicious sinks. If the malicious plugin generates new code that was not available during static AST generation, then YODA cannot access the sinks in the new code. However, we found that the tactic of function splitting to evade pattern-based detectors was more prevalent than new code generation. YODA handles function splitting by concatenate the individual function pieces defined in the AST to reconstruct the intended function. Lastly, if the plugin uses an external input for function creation (e.g., new code fetched from a URL), YODA cannot reconstruct the entire function at the AST-level.

YODA then analyzes all of the plugin code files to collect syntactic measurements, specifically: (1) the number of files and filetypes in a plugin, (2) the effective plugin state (described in §3.2.1), (3) the longest code-line length (termed $Long\_Line$) and (4) the presence of UTF-8 encoded characters or obfuscation patterns [42, 69] such as a combination of `'0'`s and `'O'`s (termed $Jumbled\_Obfus$). These measurements will be used to detect the code obfuscation behaviors in semantic analysis (Table 3.2).

Semantic Analysis. The presence of suspicious APIs alone does not equate to malicious plugin behavior. To ensure that the malicious behaviors are detected across multiple plugin files, YODA performs context-aware semantic analysis. In the dependency resolved ASTs, it marks all the sensitive APIs identified earlier as sinks and performs targeted inter-procedural backward slicing on the AST from each sink to the predefined sources using php-ast [70]. These source-sink dataflows, called *'semantic models'* are summarized in Table 3.2.

Note that YODA's models are both *composable* and *extensible*. For some dataflows, the sinks can also act as an intermediate node. For example, *Download* is a sink for the downloader malicious behavior and an intermediate node for the blackhat SEO, post injection, malvertizing, and spam injection behaviors. Since attackers extend existing techniques, this composability allows YODA to scale with evolving malware. Further, as new malware behaviors emerge (e.g., the recent trend of SSO Backdoor), analysts can easily extend YODA's models by composing existing primitive models with new API sinks used in the attack. Next, we describe each of the semantic models from Table 3.2.

1. Webshell. The plugin takes executable code as input via superglobal variables ("*Super _Global* [*input*]" in Table 3.2) which is then passed to an *Exec* sink that executes this code on the webserver.

2. Post Injection. The plugin code contains a URL that has been *Blacklist*ed as malicious by VirusTotal [71] or URLHaus [72], and it *Download*s content from this URL and inserts it as a WordPress post (*Add_Post*). We found that the URLs used by attackers are not always flagged as malicious by VirusTotal or URLHaus. We identified randomly generated strings used as throwaway domain names (e.g., `www.fatots.top`, `www.gacocs.com`) to deliver malicious content to these plugins. We provide this full list as part of the YODA source code.

3. Input Gating. Attackers protect their injected code based on a predefined

password. Here, the 'password' parameter in a super global variable ($Super\_Global\,[password]$) is set, it is conditionally evaluated, and code is executed ($Exec$) based on the conditional evaluation success. While this may appear like harmless password-protected code execution, benign plugins store client credentials in the website's database and do not employ only hard-coded passwords.

4. SSO Backdoor. Attackers are abusing the single sign on feature to create a backdoor via user accounts with admin privileges. Here, the plugin creates a user object (e.g., $Create\_User$ via `$user = array( 'user_login' => $uname,'user_pass' => $pword)`), changes the user permissions to provide administrator privileges, registers this user with the CMS (e.g. $Register\_User$ via `wp_user_insert`), and finally redirects all requests to this new user's admin URL.

5. Library Function Exists. If the plugin finds a missing library function ($Conditional \rightarrow Func\_Exists$), it locally implements the function ($Create\_Func$) to redefine it. While it is common to check if a function exists, benign plugins do not reimplement library functions but instead include the library.

6. Spam Injection. The plugin code contains a blacklisted URL (VirusTotal, URLHaus, or in-house curated URL list), and it *Download*s content from this URL and injects the downloaded content to the HTML output ($Add\_Content$) each time the website is loaded.

7. Code Obfuscation. The plugin contains (1) jumbled obfuscation patterns ($Jumbed\_Obfus$), (2) long lines of code with over 50 code instructions in the same line ($Long\_Line$) during syntactic analysis, or (3) encoded strings passed to *Decode* and *Exec* sinks. These are the 3 predominant categories of code obfuscation seen in our study. YODA could identify different obfuscation variants, and the detection module is made available as part of the YODA source code.

8. Blackhat SEO. Attackers employ conditional checks to detect if the website is

being loaded by search engine bots (SE_Bots), e.g., googlebot, bingbot, baiduspider. They $Download$ SEO campaign content from a $URL\ \ Blacklist$ and replace concealed HTML elements ($Replace\_Content$) in the plugins with this downloaded content. This impacts the website's indexing by search engines.

9. Downloader. The plugin $Download$s content from $URL\ \ Blacklist$. Note that, if YODA finds $Download$ as an intermediate sink for other attack behaviors, it assigns the appropriate attack behavior and does not flag the plugin as a downloader.

10. Function Reconstruction. To evade signature- based AVs, attackers break suspicious function names to substrings ($Str1, Str2, ..$) that can then be concatenated to form the function name. Attackers then use PHP's $Create\_Func$ to create a function that internally performs an `eval()` or executes this function.

11. Insert User. The plugin creates a user object ($Create\_User$) and registers this user account with the CMS ($Register\_User$). Benign plugins hardly add new user accounts to the CMS. Different from SSO backdoor, this user is created for one-time use and this user's contents are not loaded each time a web page is requested.

12. Malvertizing. Attackers monetize plugins to serve malicious ads. The plugin $Download$s content from a $URL\ \ Blacklist$ and redirects website visitors to a malicious site or inserts a downloaded popup ($Redirect\ \ Insert\_Popup$ in Table 3.2).

13. Fake Plugin. Attackers inject full-fledged plugins that not only give backdoor access but also run malicious code each time the website is loaded. In particular, fake plugins receive encoded payloads (generally using base64 decoding) from superglobal variables ($Super\_Global[Str]$), $Decode$ and $Exec$ this payload, and then delete it ($Delete\_Payload$).

14. Cryptominer. The plugin $Download$s a mining script from a $URL\ \ Blacklist$, writes it to a file ($File\_RW$), changes the file permission to executable, and then $Exec$s the file.

3.2.3 Origin of Malicious Plugins

YODA then determines the origin of these malicious behaviors. This helps understand the different attacker entry points within the CMS ecosystem. Our preliminary study uncovered that the malicious plugin behaviors originate from one of these four sources.

1. Nulled Plugin Marketplace. Nulled plugins commonly include multiple malicious domains (adds redundancy during domain takedown) to download malicious content on the webserver. If YODA records downloader, malvertizing, or spam injection behaviors when the effective plugin state was `'A'`, and if the plugin contains multiple redundant blacklisted URLs, it is categorized as nulled based on its behavior. Also, if the plugin name contains nulled marketplace metadata (e.g., "Shared on VestaThemes.com"), it is categorized as nulled based on its metadata. Note that not all metadata-based nulled plugins are malicious (§3.4.3).

2. Legitimate Plugin Marketplace. YODA marks the malicious origin as a legitimate plugin marketplace if: (1) one or more malicious behaviors are seen when the effective plugin state is `'A'`; or (2) the effective plugin state is `'M'` due to plugin version and/or author change[4]. Since some nulled plugins masquerade as legitimate plugins, YODA first categorizes nulled plugins with redundant malicious domains and excludes them from the legitimate marketplace category.

3. Injected Plugin. If YODA finds (1) fake plugin behavior when the effective plugin state was `'A'`, or (2) fake plugin and code obfuscation behaviors when the effective plugin state is `'M'`, the plugin is categorized as an injected plugin. Note, plugins with code obfuscation are not always injected plugins. Only if the plugin did not originate from nulled or legitimate marketplaces, then YODA marks it as an injected plugin since these plugins are not sold on marketplaces.

[4] The effective state can be `'M'` for several reasons such as code customization by the website owner, code injected by an attacker, etc. Still, the plugin version or the plugin author does not change.

4. Infected Plugin. We found that malicious plugins on the webserver tried to increase the attack's coverage by hijacking other plugins. If YODA found malicious behaviors in a plugin with effective plugin state `'M'` in an already compromised website (i.e., it has one or more malicious plugins prior to the snapshot under analysis), it is marked as infected. If it was infected when the effective plugin state was `'A'`, YODA may incorrectly label an infected plugin as originating from a legitimate marketplace. This is resolved via cross-website verification.

Cross-Website Verification. Using backups from over 400K web servers, we employ cross-website verification as an additional guarantee for the malicious origin categorization applied at a single-website level. Note, legitimate marketplace, nulled marketplace, and injected plugin categories are mutually exclusive. However, plugins from all of these categories can be infected by other malicious plugins on the webserver. YODA performs a cross-website comparison of all malicious plugins originating from legitimate or nulled marketplaces. In particular, if the identified malicious behaviors are common across all websites, then the labeled categorization is validated as correct. Otherwise, they will be correctly relabeled as infected plugins.

3.2.4 Impact Study

The origin of malicious plugins in §3.2.3 reveals the broad attacker platforms used to victimize CMS users. To understand the scale of this impact on the plugin marketplaces, YODA extracts the *impact metrics* associated with each plugin (i.e., monetary impact in terms of plugin cost and popularity impact in terms of the number of downloads) by mapping the plugins in our dataset to the plugin marketplace it originated from. We chose the 7 most popular plugin markets — three unpaid (WordPress Plugins, WordPress Themes, and Github) and four paid (ThemeForest, CodeCanyon, WPMU DEV, and Easy Digital Downloads) — to perform this study. This can be challenging because the impact metrics extraction

varies between markets due to the lack of code consistency.

To address this, we reverse-engineered the plugin-name-to-URL translation for these marketplaces, and YODA was programmed to scrape the impact metrics. YODA first constructs the URL to visit by appending the plugin name (e.g., twentytwenty) to the market-specific URL (e.g., https://wordpress.org/ themes/) and performs a GET request on the effective URL (e.g.,https://wordpress.org/themes/twentytwenty) to determine if the plugin is in the marketplace. For some marketplaces (e.g., CodeCanyon), the required URL cannot be constructed solely from the plugin name. To address this, a search query is constructed using the plugin's name. The search results are parsed to find if the target plugin exists in the marketplace. This impact metrics extraction can be extended to other marketplaces by updating YODA with the new plugin-name-to-URL translation and scraping patterns.

After obtaining the plugin's marketplace listing (i.e., a successful GET request response), the impact metrics extraction is similar across marketplaces, specific only to the web page formatting. All available metadata on the listing is stored in a database for easier queries. This metadata consists of the plugin's latest version, plugin rating, cost, and the number of sales and downloads, which when applied to a large-scale study, reveal the impact of these plugins on the community. This data will be used to infer any correlations that may exist between malicious plugins, their cost, and their popularity.

3.3 Validating YODA

To validate YODA's design considerations, we used 120 unique WordPress websites collected between Apr 2018 and Feb 2021. 60 were compromised with web-attacks as classified by pattern-based AV and the remaining 60 were randomly chosen unbiased websites. We used this dataset to establish ground truth and validate YODA's

accuracy in detecting plugins and malicious plugin behaviors on a local workstation running Ubuntu 16.04 with 32GB memory and 8 x 3.60GHz Intel Core i7 CPUs.

3.3.1 Plugin Detection Evaluation

Ground Truth. We first evaluate YODA's plugin detection. As mentioned in §3.2.1, website owners often customize their plugins, either using configuration files or explicitly modifying the PHP code. Thus, it is difficult to verify that a detected plugin matches a known plugin from the marketplace. To evaluate YODA, we need to determine if each plugin that YODA detected is either: (1) an exact match (EM), (2) a true positive match with customization (C), (3) a false positive (FP), i.e., YODA labeled a non-existent plugin, or (4) a false negative (FN), i.e., YODA missed labeling a group of files as a plugin.

We used YODA to identify an initial list of plugins in all 120 website backups. This list contained EMs, Cs, or FPs (per above). To determine which, we created a ground truth plugin set by downloading these plugins from the plugin marketplaces. We also contacted the authors of paid plugins found in our dataset and received all versions of these plugins as well. We compared all the files (via MD5 hash) for each plugin detected by YODA against the files for the *same version* of the plugin within the ground truth set. If 100% of the files from the ground truth plugin matched those in the detected plugin, we classify it as an EM (Table 3.4 Column 3). Greater than 90% match is considered a true positive with customization (C, Column 4). If the comparison led to a less than 90% match, we classified this plugin as an FP (Column 5). In fact, customized plugins rarely differ by more than one file (we found only 8 instances of multi-file customization in our dataset), thus a 90% match is so strict that it is less favorable to YODA, but we aim to aggressively flag any FPs. We manually investigated all mismatches.

To check for the FNs, we pulled every version of all freely available plugins from

Total #Websites: 120	Total #Plugins: 3,168						
Plugin Name	#Y	TP		FP	FN[3]	LV_m	LV_M
		EM[1]	C[2]				
Yoast SEO	47	27	18	2	0	26	38
Contact Form 7	41	28	12	1	0	23	28
Wordfence Security	37	30	6	1	0	3	26
Manage WP - Worker	34	20	14	0	0	14	23
Add From Server	31	19	12	0	0	16	24
Shield Security	29	12	16	1	0	4	13
WP Rocket	27	10	15	2	0	10	12
MainWP Child	25	18	6	1	1	11	19
Easy WP SMTP	24	12	12	0	0	9	13
Amazon Web Services	20	18	2	0	0	6	17
Simple Social Icons	19	19	0	0	0	15	15
WP Offload S3 Lite	18	17	0	1	0	7	14
Jetpack	17	10	6	1	0	2	12
Sharedaddy	15	12	2	1	0	13	13
Akismet Anti-Spam	14	13	1	0	0	1	11
Total Top 15	398	265	122	11	1	160	278
Total Overall	3,168	2,240	889	39	3	728	2,060

1: #W where the plugin exactly matches the ground truth plugin.

2: #W with customized plugins correctly identified by YODA.

3: #W with customized plugins incorrectly identified by YODA.

Table 3.4: Plugin Detection Evaluation.

the WordPress SVN[5] plugin repository. We also added all versions of the free and paid plugins from above. We then compared all file hashes from the downloaded plugins against the files in each website. If 90% or more of the plugin's files match files in the website and YODA did not mark the group of files as a plugin, we count this is as an FN.

Detection Results. In the 120 websites, YODA found a total of 3,168 plugin instances (#Y). Table 3.4 summarizes the results for all plugins and drills down into these results for the top 15 plugins based on their popularity in our dataset. Of the 3,168 plugin instances in Table 3.4, YODA correctly detected 3,129 (i.e., 2,240 + 889) plugins. 2,240 of these plugins exactly matched the ground truth dataset, and

[5]WordPress uses SVN to maintain version-controlled plugins.

889 plugins were customized — a TP rate of 98.7%.

We manually verified all the plugins marked as FP and found that only 11 of the 39 plugins were actually FPs (i.e., a group of files incorrectly identified as a plugin). Here, the website owner copied the plugin root file (containing plugin APIs and missing referenced files) to their home directory (likely part of customization or backup), misleading YODA into identifying a group of files as a plugin. The remaining 28 of the 39 plugins either redefined the base WordPress APIs or replaced them entirely with custom APIs; such heavy customization in a single file put them below the strict 90% match. Thus, despite using a strict measure for FPs, the FP rate is reasonable (1.2%). The 3 FNs we found were due to the website owners deleting the plugin header as part of customization. Since the header holds the highest weight for determining a plugin group, YODA missed identifying these plugins.

We now use this dataset and the plugin version extracted by YODA to understand if CMS-users keep their plugins updated to the latest version. Columns 7 and 8 show the number of websites that had the plugins at the latest minor version (LV_m) and the latest major version (LV_M). For example, if the latest plugin version on the marketplace is 4.3.6 and it matches our dataset plugin version, we count it as LV_m. If our dataset plugin version is 4.3.2, since the major version (i.e., 4.3) is still up to date, we count this as LV_M.[6] Only 40% (160 of 398) of the top 15 plugins and 23% (728 of 3,168) of all plugins were updated to the latest minor version. From Column 8, we find that about 70% (278 of 398) of the top 15 plugins and 65% (2,060 of 3,168) of all plugins are updated to the latest major version. Over 35% of all plugins used are clearly outdated.

3.3.2 Malicious Behavior Evaluation

Ground Truth. After establishing confidence in YODA's plugin detection, we now evaluate the accuracy of identifying malicious plugins. We eliminated the 39 FP

[6] Latest major version includes all latest minor versions.

Total #Websites: 120	Total #Plugins: 3,132					
Malicious Behavior	#W	#GT	#Y	TP	FP	FP
Code Obfuscation	15	28	28	28	0	0
Webshell	19	23	26	23	3	0
Function Reconstruction	7	16	18	16	2	0
Downloader	7	12	14	12	2	0
Library Function Exists	10	13	14	13	1	0
Input Gating	4	13	13	13	0	0
Fake Plugin	3	7	7	7	0	0
Spam Injection	3	6	6	6	0	0
Malvertising	2	5	5	5	0	0
Insert User	2	3	3	3	0	0
Blackhat SEO	1	2	2	2	0	0
Post Injection	1	2	2	2	0	0
SSO Backdoor	1	2	2	2	0	0
Cryptominer	1	1	1	1	0	0
Total Malicious Plugins[1]	61	84	89	84	5	0
Total Benign Plugins	120	3,048	3,043	3,043	0	5

1: This is not the sum of the columns, but the total #websites and #plugins in the evaluation dataset with malicious behaviors.

Table 3.5: Evaluation of the Malicious Behavior Detection.

and included the 3 FN plugins from the same 3,168 plugins from above, and our team manually verified the server-side code files in all 3,132 plugins and tagged them with corresponding malicious behavior labels.[7] We then ran YODA's malicious behavior detection on all of these plugins and compared the labels assigned by YODA with the manually derived labels. The results are presented in Table 3.5.

Detection Results. In our dataset of 3,168 plugins across 120 websites, YODA reported 61 websites (#W) containing 89 plugin instances (#Y) that exhibit malicious behaviors whereas our manually labelled ground truth (#GT) showed that only 84 plugins across these websites were malicious. Recall, our dataset has 60 websites with known-compromises (i.e., web attacks detected), and 58 of these websites contained malicious plugins. In addition, YODA found 3 websites containing malicious plugins in the 60 randomly chosen websites. The malicious behaviors reported by YODA

[7]YODA did not have access to our manually derived labels.

matched our ground truth for plugins from these 61 websites (i.e., TP). Based on our manual verification, we did not find any plugins that contained malicious behaviors missed by YODA, thus showing zero FNs.

Table 3.5 shows YODA produced FP detections for 8 behavior instances in 5 plugins. Our manual investigation revealed that 4 of these plugins used a combination of behaviors that resembled webshells (i.e., executing decoded content) and checking if the library function `base64_decode` exists and redefining it if not. Our investigation confirmed that these plugins did not show any outright malicious activity, but this rarely-benign code implementation misled YODA. Also, 2 plugins were falsely labeled as downloaders, due to VirusTotal falsely blacklisting the extracted URLs as malicious. This gives us confidence that YODA accurately detects plugins and malicious behaviors. Table 3.5 also summarizes the benign plugins in this dataset that we verified were not malicious.

3.4 Deploying YODA

#Websites	410,122		
Min. Duration	102 days	Min. #Plugins	1
Avg. Duration	406 days	Avg. #Plugins	49
Max. Duration	3,259 days	Max. #Plugins	68

Table 3.6: Dataset Summary.

We deployed YODA on the full dataset of 410,122 unique WordPress websites' nightly backups (Table 3.6). This dataset provides a realistic view of the plugin ecosystem because over 37% of the world's websites and over 63% of CMS-based websites run on WordPress [18]. It also allows us to retroactively deploy YODA over 8 years. The backups contain an average of 406 day-snapshots per website. Many backups went all the way back to 2012, representing some of the earliest customers of CodeGuard. Each website had between 1-68 plugins, with an average of 49 plugins per website. This high average shows that most website owners place *unwarranted*

Malicious Behavior	#W	#P	IR[1]	First Seen	Temporal Evolution (07-2012 - 07-2020)	Marketplace		Injected		Nulled		Infected	
						#W	#P	#W	#P	#W	#P	#W	#P
Webshell	7,921	10,279	1.3	Jul 2012		10	12	854	994	160	232	7,117	9,943
Code Obf	6,752	10,064	1.5	Aug 2012		0	0	409	558	1,055	1,214	5,509	8,819
Input Gating	5,928	23,140	3.9	Jul 2012		0	0	47	50	3,445	7,821	5,588	20,684
Downloader	2,314	5,944	3.6	Mar 2014		151	288	19	20	1,540	2,683	1,562	4,254
Spam Injection	1,202	3,723	3.1	Oct 2016		1,166	3,452	0	0	0	0	36	271
Lib Func Exists	2,233	3,576	1.6	Aug 2012		25	29	5	5	154	241	2,195	3,475
Blackhat SEO	1,358	1,714	1.3	Oct 2013		86	86	8	21	534	650	857	1,421
Fake Plugin	1,121	1,336	1.2	Jul 2014		0	0	1,121	1,336	0	0	0	0
Func Reconst	636	929	1.5	Jan 2016		3	3	52	54	12	13	579	890
Insert User	357	1,531	4.3	Dec 2015		0	0	266	266	2	6	292	1,490
Post Injection	281	1,407	5.0	May 2016		0	0	266	266	1	1	315	1,415
Malvertising	915	1,354	1.5	May 2017		12	13	0	0	894	1,330	13	13
SSO Backdoor	191	905	4.7	May 2019		2	2	0	0	36	91	190	879
Cryptominer	4	4	1.0	Jul 2018		0	0	4	4	0	0	0	0
Total[2]	24,931	47,337	1.9	Jul 2012		1,345	3,685	1,201	2,814	5,244	8,525	18,034	40,533

1: Infection Ratio (IR) is the ratio of #P to #W, shows a measure of infection spread.

2: This is not the sum of the columns, but the total #websites and #plugins with malicious behaviors in our dataset.

Table 3.7: Distribution and Temporal Evolution of the Malicious Behaviors Across all Websites in our Dataset.

trust in plugins to keep their websites up and running.

Experimental Setup. We used Amazon Web Services (AWS) Elastic Compute (EC2) r5.2xlarge instances with 8 virtual CPUs and 64 GB of RAM to run YODA on the website backups. These instances were supervised by the AWS Batch job scheduling engine to deploy YODA on hundreds of backups in parallel.

3.4.1 Malicious Behavior Evolution

YODA found malicious plugin instances (#P) in 24,931 of the 410,122 websites (#W), shown in Table 3.7. As expected, over 10K malicious plugin instances used the age-old web attack techniques: webshells and code obfuscation. The infection ratio (IR, the ratio of #P to #W) shows a measure of infection spread. Several malicious behaviors have IR >3, implying that multiple plugins within the same website contain these same malicious behaviors. Closer inspection revealed that these are due to *plugin-to-plugin infection*: a single malicious plugin on the webserver infects multiple benign plugins, replicating the behavior.

Dating back to 2012, we studied the evolution of these malicious behaviors. Since the absolute number of websites in our dataset increased over time, in Table 3.7 Temporal Evolution, we plot the newly infected websites as a percentage of all malicious websites to remove dataset bias. While some attack behaviors were popular since late 2012, other behaviors such as spam injection (2016), malvertising (2017), and SSO backdoor (2019) were introduced recently. However, it is interesting to note that regardless of when they were first introduced, all of these behaviors are still prevalent in present-day malicious plugins. A closer look at the absolute values of the newly introduced malicious behaviors reveals that the number of malicious plugins peaked in March 2020, which notably coincides with the COVID-19 outbreak.

Thousands of malicious plugins originated from legitimate plugin marketplaces. Table 3.7's Marketplace Columns show their distribution (i.e., number of websites

#W and number of plugins #P with malicious behaviors). Row 2 shows that *none of these plugins use code obfuscation techniques* — despite being sold on legitimate marketplaces they brazenly hide in plain sight. Attackers (rightly) assume that an average website owner will not inspect the plugin code before installing it on their webserver. In fact, we found instances of well commented malicious code in 2,379 of the 3,452 plugins that performed spam injection originating from legitimate plugin marketplaces. Evidently, these plugins enabled illegal monetization via blackhat SEO, downloader, and spam injection in 86, 288, and 3,452 plugin instances, respectively.

Attackers exploited the scalable CMS infrastructure to inject malicious plugins into websites. Table 3.7's Injected Columns show that the injected plugins aim to gain and maintain access to the webserver. They are injected without the website owners' knowledge and over 80% of these plugins had fake plugin behaviors (1,336), webshells (994), or obfuscated code (558). Although cryptomining is gaining popularity, we only found 4 injected cryptominer plugins on 4 websites revealing its infancy in pervading the CMS landscape.

We found 8,525 malicious nulled plugin instances in our dataset that exploit human vulnerabilities to rapidly spread malware. Table 3.7's Nulled Columns show that over 91% (7,821 of 8,525) of these plugin instances used input gating (i.e., password-protecting the publicly accessible code) to thwart competing attackers from introducing malicious payloads. We also found that the plugins introduced after December 2018 primarily employed downloader, blackhat SEO, and malvertizing behaviors in 2,683, 650, and 1,330 plugin instances, respectively, to infect other benign plugins.

It was concerning that over 40K plugin instances were infected post-deployment. Table 3.7's Infected Columns show that these plugins portray a variety of malicious behaviors. Most attackers employ behaviors such as webshells, obfuscation, and downloaders in 9,943, 8,819, and 4,254 plugin instances, respectively. Interestingly,

over 50% (20,684 of 40,533) of these plugins employed input gating showing attackers' diligence in marking their conquered territories.

3.4.2 Fueling the Malware Economy

Markeplace	Malicious			Downloads Range			Cost
	#P	#U[1]	%M[2]	Min.	Avg.	Max.	
Legitimate Marketplace							
WP Themes	523	62	1.1%	7.7K	336K	3.6M	-
WP Plugins	1,583	69	0.25%	4	945K	25M	-
Github	0	0	0%	-	-	-	-
WPMU DEV	132	2	1.8%	54K	510K	524K	$25.8K
CodeCanyon	164	10	0.4%	1	40	73	$6.8K
ThemeForest	195	22	0.37%	9	20K	213K	$8.9K
EDD	0	0	0%	-	-	-	$0
Subtotal	2,597	165	0.38%	-	-	-	**$41.5K**
Nulled Plugins							
WP Themes	1,074	59	1.08%	11K	203K	5.7M	-
WP Plugins	146	43	0.16%	65	4K	37K	-
Github	0	0	0%	-	-	-	-
WPMU DEV	4	1	0.9%	572K	572K	572K	$2.3K
CodeCanyon	2,085	122	5.02%	1	70	570	$82.3K
ThemeForest	3,059	223	3.82%	3	12K	213K	$142K
EDD	39	3	1.2%	-	-	-	$1.3K
Subtotal	6,407	451	1.03%	-	-	-	**$228K**
Infected Plugins							
WP Themes	9,776	1,864	34.2%	1	367K	7.4M	-
WP Plugins	8,049	6,520	23.8%	2	4M	260M	-
Github	15	1	0.01%	2	2	2	-
WPMU DEV	450	9	8.2%	187K	2M	10.5M	$88.2K
CodeCanyon	1,873	469	19.3%	1	62	563	$59.9K
ThemeForest	5,858	1,072	18.4%	2	10K	213K	$264K
EDD	634	57	23.3%	-	-	-	$422K
Subtotal	26,655	9,992	22.9%	-	-	-	**$834K**

1: #U: Number of unique malicious plugins, 2: %M: Percentage of the plugins on the marketplace that were flagged as malicious

Table 3.8: The Economy of Malicious Plugin Marketplaces.

Next, we turned our attention to the economic drivers of these malicious plugins. Table 3.8 categorizes our results based on the origin of malicious behaviors, i.e., legitimate marketplaces, nulled marketplaces, and infected plugins.

Table 3.8 begins with malicious plugins originating from legitimate plugin marketplaces. About 70% of these (2,597 of 3,685) were found on 5 of the 7 most popular marketplaces. Our dataset alone constituted over $41K in purchases of malicious plugins from legitimate marketplaces. We found 62 unique malicious plugins from WP Themes and 69 from WP Plugins (unpaid marketplaces), contributing to 1.1% and 0.25% of these marketplaces, respectively (%M Column). Furthermore, the malicious plugins from these marketplaces are extremely popular, averaging 336K and 945K downloads per plugin. We also found 34 unique malicious plugins sold on paid marketplaces.

Nulled plugins impersonate plugins from legitimate marketplaces. YODA extracts their popularity and cost from legitimate marketplaces. The cost represents the explicit losses incurred by the legitimate plugin authors. About 75% of the malicious nulled plugins (i.e., 6,407 of 8,525) in our dataset contain legitimate counterparts in these 7 popular marketplaces. Since nulled marketplaces distribute plugins free of cost, we did not expect to find plugins from unpaid marketplaces. Surprisingly, we found a total of 102 plugins from WP Plugins and WP Themes sold on nulled marketplaces. As expected, we also found that over 77% (349 of 451) of the nulled plugin counterparts were sold on paid marketplaces. Attackers impersonated 122 and 223 best-selling plugins from CodeCanyon and ThemeForest, respectively. Overall, the website owners from our dataset alone contributed to $228K in explicit loss to the plugin authors. This shows that attackers are successfully targeting psychological human vulnerabilities and the less-technical CMS users are installing pirated plugins.

Finally, Table 3.8 considers the origin of post-deployment infected plugins. About 65% of the infected plugins (i.e., 26,655 of 40,533) were downloaded from these 7 popular marketplaces. Since the plugins from WP Plugins and WP Themes are widely used, they are also commonly infected. 34.2% and 23.8% of plugins from

WP Themes and WP Plugins became victims of plugin infections. Despite paying a premium for plugins from paid marketplaces, a significant number of these were found to be infected, i.e., 8.2% of WPMU DEV, 19.3% of CodeCanyon, 18.4% of ThemeForest, and 23.3% of EDD. The website owners spent a total of $834K on these plugins, only to find them compromised. This encapsulates the additional implicit cost of malware cleanup incurred by installing malicious plugins from legitimate and nulled marketplaces.

3.4.3 Nulled Marketplace Study

Since nulled plugins require some tampering with the WordPress backend (too complex for the typical CMS user), we did not expect to find many nulled plugin instances in our dataset. Surprisingly, Table 3.9 reveals 6,223 websites had at least one nulled plugin. We found that these plugins are gaining popularity by optimizing for search engine ranking.

A Google search for any "_____ WordPress plugin/theme free download" almost always has a nulled marketplace in the top five results. Figure 3.4 shows the search engine results for one of the popular WordPress themes, DooPlay, normally priced at $80 [80]. Here, the highlighted four of the top five results on Google search lead us to nulled marketplaces that are known for distributing malicious plugins and themes. In Nov 2019, WordFence alerted the community about a rouge blackhat SEO malware campaign via nulled malicious plugins and themes [48]. Despite this knowledge, the attackers have successfully maintained their ranks on the search engine results.

Table 3.9 shows the nulled marketplaces extracted from the plugin metadata. If YODA identifies a plugin as nulled based on its behavior alone and if it cannot extract a nulled marketplace from the plugin metadata, we categorize the marketplace as 'Unknown'. Table 3.9 shows `vestathemes.com` as the most popular nulled plugin marketplace in our dataset, with 3,177 plugin instances (#P) downloaded across 2,398 websites (#W). Recall from §3.2.3, not all nulled plugins portray malicious behaviors.

Nulled Marketplace	#W	#P	#MW[1]	#MP[2]	%M	1st Seen	1st Mal. Seen	Popular Mal. Plg.	Cost	#Instances
vestathemes.com	2,398	3,177	2,283	3,057	96.2%	Aug 2014	Jul 2018	Flatsome [73]	$59	195
www.themes24x7.com	989	1,363	928	1,282	94.1%	Mar 2016	Mar 2016	WPBakery Page Builder [74]	$64	140
www.wplocker.com	841	1,035	829	1,017	98.3%	Nov 2013	Jan 2014	FormCraft [75]	$36	80
www.jojo-themes.net	133	141	109	117	82.9%	Feb 2016	Feb 2016	Gravity Forms [76]	$59	9
theme123.net	127	144	127	144	100%	Dec 2013	Dec 2013	WP Robot 5 [77]	$89	9
mafiashare.net	121	149	117	142	95.3%	Jan 2014	Dec 2015	BeTheme [78]	$59	9
www.wptry.org	79	99	79	98	98.9%	Apr 2020	Apr 2020	Woodmart [79]	$59	3
themlot.net	60	65	60	65	100%	Dec 2014	Dec 2014	BeTheme [78]	$59	4
Unknown	1,906	2,603	1,906	2,603	100%	Oct 2016	Oct 2016	Flatsome [73]	$59	252
Total[1]	6,223	8,776	5,244	8,525	97.1%	Nov 2013	Dec 2013	Flatsome [73]	$59	483

1: #MW: The number of websites with malicious nulled plugins, 2: #MP: The number of malicious nulled plugins.

3: The total here is not the sum of the columns, but the total #W and #P from nulled marketplaces in our dataset.

Table 3.9: Study of Malicious Plugins From Nulled Marketplaces.

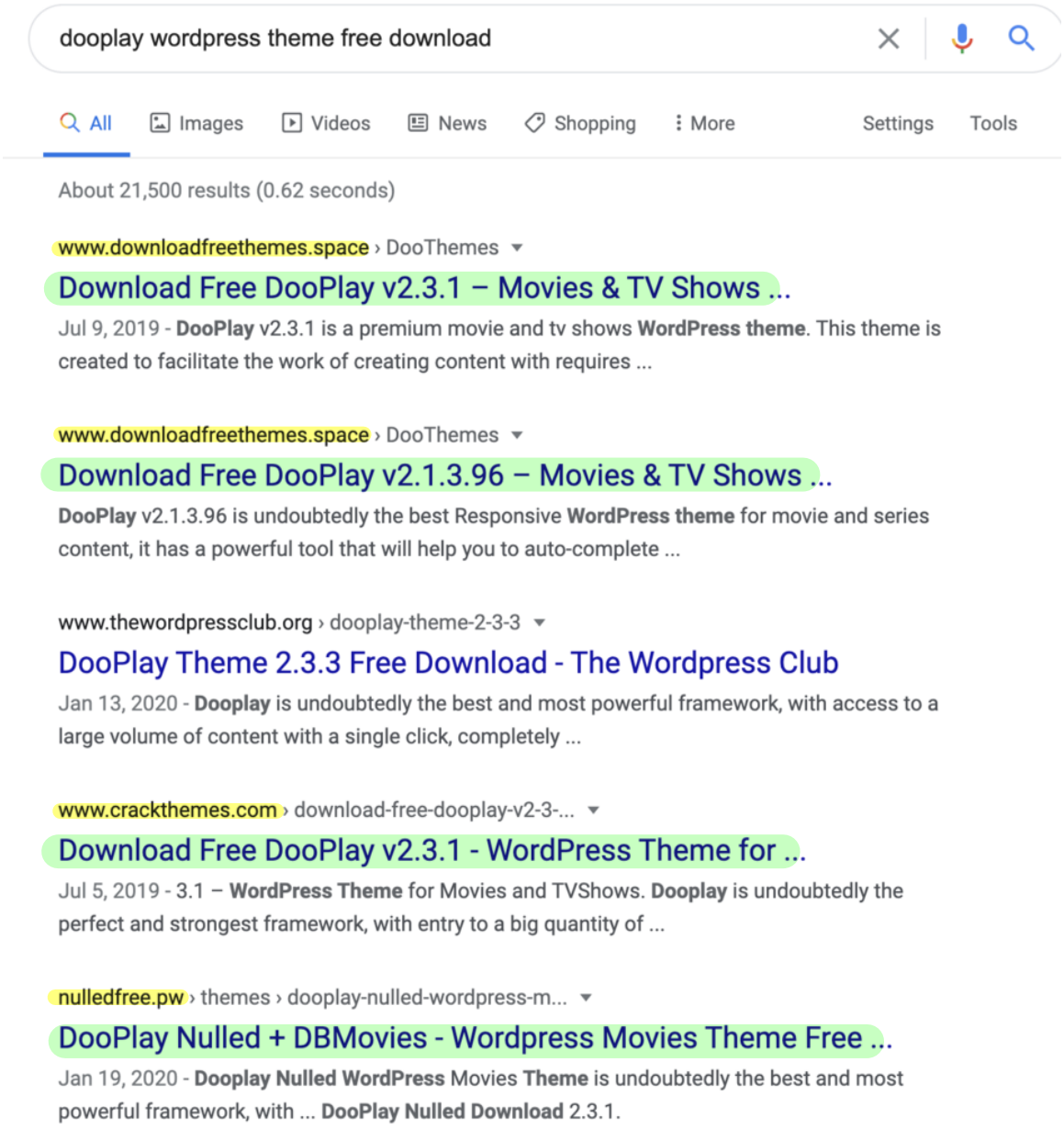

Figure 3.4: Google Search Results of a Typical Paid Plugin.

Columns 4-5 present the number of websites (#MW) containing malicious nulled plugins (#MP). Overall, over 97% of all nulled plugins deliver malicious behaviors (%M). In particular, 100% of the plugins we saw from `theme123.net`, `themelot.net`, and 'Unknown' marketplace were malicious.

Interestingly, the 'Unknown' marketplaces have distributed over 31% of all malicious nulled plugins (2,603 of 8,525) in our dataset. They impersonate the

plugin author entirely and hide that they were downloaded from a nulled source as opposed to the other nulled marketplaces in Column 1. A comparison of the plugin header from an 'Unknown' nulled plugin and a legitimate marketplace plugin did not reveal any differences. Only after matching the code files were we able to tell a nulled malicious plugin apart from the legitimate plugin. However, since the website owners cannot compare the nulled plugin's code to the paid legitimate plugin, it is impossible for them to identify the nulled plugin as malicious. However, YODA can detect malicious plugins by only analyzing its code files.

Table 3.9 also shows that most nulled marketplaces have been around for a long time, since 2013-2014. However, over 50% of the marketplaces displayed malicious behaviors starting in 2016. In particular, the 'Unknown' marketplaces have attempted to spread malware since 2016 and have been successful through 2020. The most popular nulled plugins cost between $36 and $89, with an average of $59 per plugin. 447 of the 483 popular malicious nulled plugin instances were Flatsome [73], a WordPress theme that would normally cost the website owner $59 and provided pre-defined layouts for user-friendly e-commerce shop features.

3.4.4 Are Infected Plugins Cleaned Up?

Lastly, Table 3.10 studies the plugin clean-up statistics to understand how attackers are evading website owners. Very few website owners (2,697 of 24,931 or 10.8% of the compromised websites overall) attempt to clean up the malicious plugins on their webserver. We hypothesize that those website owners are unaware of the malicious plugins or they cannot correlate malware side-effects (such as server slowdown) with the plugins. As seen in Table 3.10, 24.1% of websites with malicious plugins from legitimate marketplaces are cleaned up, the highest rate by far. Only 6.7% of nulled plugins are cleaned up, which further strengthens our hypothesis (§3.5) that despite much later adoption, nulled plugins provide robust persistence for attackers.

Of the 2,697 websites that attempted to clean up 7,042 malicious plugins, 12.5%

Malicious Origin	Malicious		Cleaned Up			Reinfected			Still Infected		
	#W	#P	#W	% W	#P	#W	% W	#P	#W	% W	#P
Marketplace	1,345	3,685	324	24.1%	389	32	9.9%	32	1,090	81.0%	3,327
Injected	1,201	2,814	169	14.1%	221	21	12.4%	41	1,067	88.8%	2,608
Nulled	5,244	8,525	353	6.7%	471	63	17.8%	81	5,003	95.4%	8,115
Infected	18,034	40,533	2,174	12.1%	5,962	254	11.7%	551	16,881	93.6%	34,956
Total[1]	24,931	47,337	2,697	10.8%	7,042	336	12.5%	705	23,577	94.6%	40,787

1: The total here is not the sum of the columns, but the total #W and #P in our dataset.

Table 3.10: The Cleanup and Reinfection Distribution of Malicious Plugins.

of the websites (336 of 2,697) were reinfected. Interestingly, nulled plugins were most consistently reinfected (17.8% or 63 of 353 websites). Plugins downloaded from legitimate marketplaces show the least rate of reinfection (9.9%). This can be attributed to community engagement in identifying malicious plugins on legitimate marketplaces. Such plugins are either purged from the marketplace or their authors are forced to remove the malicious code.

Lastly, we measured the websites that remained infected up to the time of writing. Despite cleanup efforts, over 94% of all websites with malicious plugins remained infected. This proves that CMS plugins have provided a reliable webserver infiltration vector for nearly a decade.

3.5 Persistence of Malicious Plugins

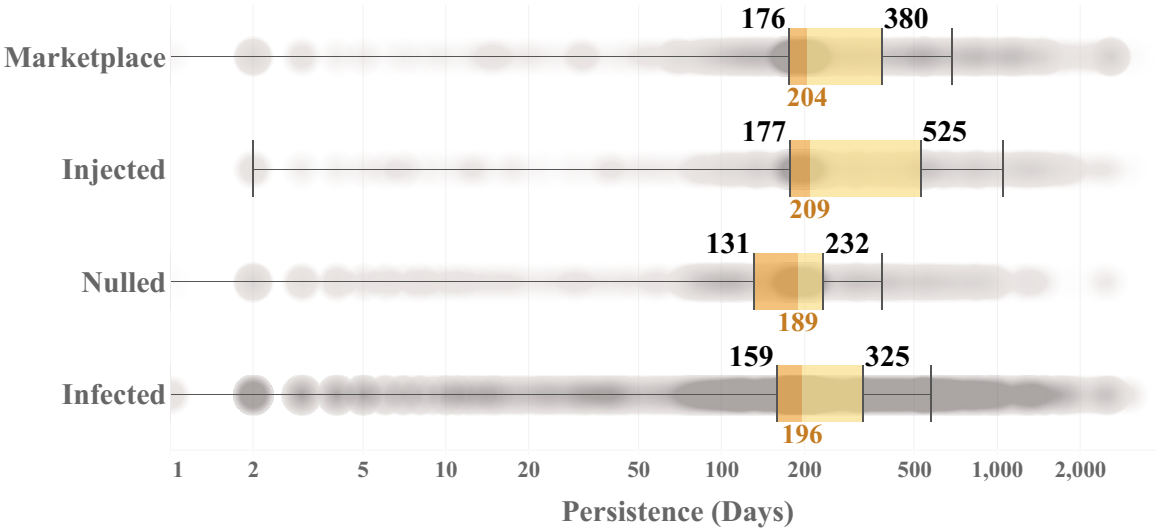

Figure 3.5: Persistence of Malicious Plugins.

To understand the persistence patterns of malicious plugins, Figure 3.5 shows a box plot measuring the number of days malicious plugins were identified on the webserver, categorized by their origin. The median persistence ranges between 189-209 days. Thus, over 50% of the malicious plugins persist for over 6 months. We also noted that over 80% of the remaining malicious plugins (those that persisted for less than 6 months) were introduced during Feb - Mar 2020 and persisted through the

end of our study. This confirms our previous observation (Table 3.10) that 94% of the malicious plugins in our dataset installed over 8 years are still active today.

Popular plugins on legitimate marketplaces mostly introduce malicious behaviors via plugin updates. Thus, we assumed that these behaviors would be cleaned up with updates[8] as well. As seen from Figure 3.5, malicious plugins from legitimate marketplaces are not immediately identified at source and persist for 176 - 380 days. Recall from §3.3.1, over 60% of the website owners do not enable auto-updates and use outdated plugin versions. If these website owners happen to install a malicious version of a plugin from a legitimate marketplace, it persists for months or years.

Figure 3.5 also shows that the persistence of nulled plugins (131 - 232 days) is shorter compared to other origins. This can be attributed to the fact that even though nulled marketplaces existed since 2013, they gained popularity around 2018, and their blackhat SEO campaigns accelerated in early 2019. We found that once nulled plugins are installed on the webserver, they are rarely removed (§3.4.4). The website remains compromised since the website owner is unaware of the plugin's malicious intentions. So despite much later adoption, 25% of these plugins persist for over 232 days.

Notably, it is the injected plugins that win the persistence war. Over 75% of these plugins remain active for at least 177 days, and over 25% of these plugins persist for at least 525 days. This proves that injected plugins are never noticed by the website owners, who typically use GUIs to manage their CMS.

3.6 Case Studies

1. Malvertizing URLs. Discovered in 2019, the largest known malvertizing campaign downloaded content from malicious domains in plugins to the webserver [48]. To understand the lifecycle of these domains, we extracted 352 URLs from all malvertizing plugins in our dataset and analyzed the domain creation

[8] The marketplace takes down community-identified malicious plugins or mandates reverting the malicious behaviors.

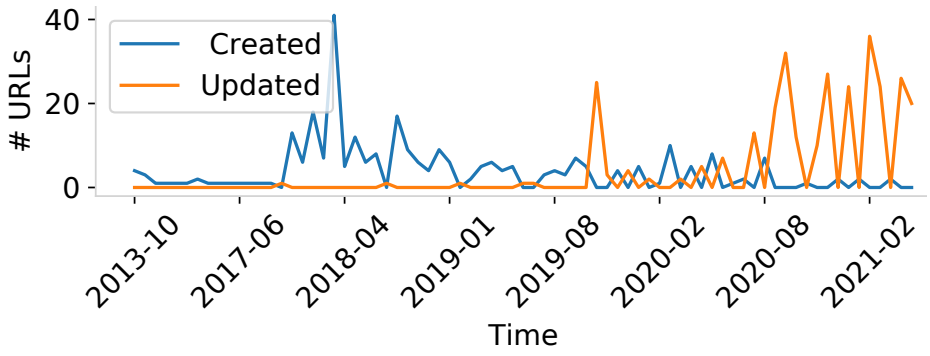

Figure 3.6: Malicious URLs Created and Updated.

date, last updated date (i.e., registration renewed), and their registrars. Figure 3.6 shows the distribution of URLs created and updated over time. The majority of these URLs were created in 2018, but attackers are re-registering the same URLs with peak activity in 2021. Thus, the malvertizing campaign is still active (confirmed by their use in recent malicious plugins) and has evaded detection. In fact, only 56 of these URLs were no longer registered at the time of writing. We believe that these were throwaway URLs generated for a short targeted attack. We also found 38 of these URLs captured by the internet archive [81], further supporting our hypothesis.

2. Spam Injection Insights. Starting in 2016, a prolific spammer bought and updated several WordPress plugins for a coordinated spam campaign over a 4.5-year period [47]. Hoping to find how widespread this campaign was among our dataset, we drilled down into the spam injection plugins YODA detected. Apart from downloading malicious spam content from the spammer's own domains onto the webserver, we discovered these same plugins also collected details on visitors to the infected website, such as URL, IP, user agent, and other attacker-defined variables. Of the 3,723 spam injection plugins, 94% sent back IP and user agent using the PHP superglobal variables. Of these, 66% encoded the IP using `$_SERVER['REMOTE_ADDR']` and 34% used `$_SERVER[ 'SERVER_ADDR']`. These plugins also sent attacker- defined variables `'p'` (set to 2, 29, or 9) and `'v'` (set to

11 or 18). While we cannot accurately decipher what these variables mean, we speculate that they identify the spammer's campaign and distribute profits similar to affiliate tracking. 6% of the plugins did not send any data back to the attacker. Interestingly, these where all the earliest cases that appeared in late 2016.

3.7 Limitations and Future Work

Additional CMS Platforms. YODA can accurately detect malicious plugins on WordPress-based websites. However, scaling to other CMSs only requires updating YODA's plugin detection and semantic models. YODA's modularity enables porting to other platforms by reviewing the API documentation of the target CMS. We leave handling other CMSs as future work.

Static Behavior Detection. Since YODA relies on static analysis, it carries the limitations of static analysis. Like any static data flow analysis framework, YODA can identify obfuscated code but it cannot detect malicious behaviors within the obfuscated code. YODA could be augmented with dynamic analysis [82, 69] to achieve better coverage of dynamic PHP code.

Semantic Model Evasion. Since the semantic models rely on data flows, they cannot be evaded by rearranging code, inserting junk code, or splitting the attack behavior across files. That said, attackers can try to evade YODA in two ways: (1) evolve to entirely new behaviors (e.g., the recently introduced SSO Backdoor behavior) or (2) novel implementations of known attack behaviors (via new PHP APIs). Such evolution is expected, and in both cases, new semantic models can be crafted for the new data source-sink combinations.

CHAPTER 4

THE MALWARE THAT KEEPS ON GIVING: A DECADE-LONG STUDY OF OBFUSCATION AND PACKING ON SERVER-SIDE WEB MALWARE

Most modern websites are built on content management systems (CMS) by non-technical users [83]. Recent research reported that attackers are exploiting these websites at scale by carelessly dropping thousands of obfuscated and packed malicious files on the webserver [42, 84]. This is counter-intuitive, since attackers are assumed to be stealthy. However, our study found that oversights in our existing defenses in the domain of exploited webservers have *made the attackers lazy*. Put simply, weak defenses are making attackers respond with simple and incremental offenses. In fact, our study revealed that out of 9.2M unpacked malware payloads captured from exploited webservers since 2020 *only 29% were previously unknown*, and the rest were naively repacked known malware.

Several research solutions analyzed packing in client-side web malware (i.e., executes in the visitor's browser) via static [85, 86] and dynamic analysis [87, 88, 89, 90]. On the contrary, little research has been done on packing and obfuscation schemes in server-side web malware (i.e., they execute on the exploited webserver) despite their increasing presence. In fact, several studies acknowledged the presence of obfuscation in exploited production websites [42, 84, 91]. However, due to the limitations of server-side malware analysis, packing/ obfuscation analysis was left for future work. Starov et. al. [92] focused on studying packed server-side webshells alone using honeypots for data collection and relied on UnPHP [93], a commercial PHP unpacker, to unpack and deobfuscate packed malware. We found (in §4.3.2) that UnPHP does not keep up with evolving malware packing techniques. None of

these works studied how obfuscation and packing techniques evolved over time for server-side web malware.

Most modern website owners are less-technical and the stakes for securing their websites are low [42]. As such, they are either unaware of the thousands of dropped packed malware, do not have the resources to fix the compromise, or are unmotivated to take action. The webserver security industry has begun selling products that rely on pattern-matching, entropy analysis, or static analysis at best [94]. To make matters worse, attackers frequently inject packed malware payloads into benign code files within the CMS framework [95]. This increases the complexity of analysis for defenders who must also distinguish malware from benign code. In §4.5.1, we compare packed server-side malware to traditional Windows malware packing schemes and find that server-side dynamic code generation enables superior polymorphism that can evade the existing defenses [96, 91].

This research had the unique opportunity to collaborate with CodeGuard, one of the largest website backup service providers on the market, who furnished us with access to the nightly backup snapshots of over 500K unique production websites maintained since 2012. We began by assessing a subset of 85 websites from this dataset that our collaborator identified as compromised. Our preliminary investigation revealed that 65 of these websites contained packed files. Further analysis of these packed files revealed something we had not expected: attackers did not use sophisticated techniques such as multi-threading, environmental awareness tests, nested conditional branches, confusing automated analyses, and timing-based evasion. Instead, they relied on simple ciphers and encoding schemes coupled with the ever- present dynamic code generation to create polymorphic variants of the same payload year after year.

Webserver security research and practice is essentially stuck in the past (confounded by re-packed old malware) without an actionable understanding of

obfuscated and packed server-side malware. To this end, we developed OBIWAN, an automated dynamic analysis-based deobfuscation and unpacking framework that unpacks code layers to reveal the malware payloads, which can then be passed to existing detection systems. When deployed at scale, OBIWAN explicitly helps the website owners, hosting providers, and backup service providers identify packed malware on their servers as well as commercial anti-viruses (AV) to improve their server-side malware detection. It also implicitly helps search engines and advertising agencies since most server-side web malware on compromised websites perform blackhat SEO and ad frauds [97].

We deployed OBIWAN on CodeGuard's dataset and performed a large-scale retroactive decade-long study of obfuscated and packed server-side web malware. This study found over 10.1M obfuscated malware across over 27K websites, and 8.7M of these malware used packing. Our research highlighted that packing enables attackers to evade malware detection systems successfully. In fact, 63% of the deepest unpacked layers of the malware payload were identified as malicious by VirusTotal [98]. Our research also revealed that commercial server-side unpacking tools cannot reliably unpack malware. UnPHP [93] could only unpack 14.5% of the obfuscated malware unpacked by OBIWAN. We hope that the insights from this study can help us improve defenses and put up a stronger fight against webserver malware. We will release a representative subset of unique obfuscated files identified in our dataset after anonymizing any PII.

4.1 Background

To derive the types of obfuscation and packing complexities of server-side web malware, we performed a preliminary study by analyzing obfuscated and packed web malware. To this end, we chose 65 known-compromised website backups identified by CodeGuard as our preliminary analysis dataset. We identified

obfuscated files on these websites using techniques proposed by past work [42, 91]. We found that these websites had a total of 6,939 obfuscated server-side code files and 826 unique files. We manually analyzed these 826 files to understand common obfuscation techniques used by attackers. We then tried unpacking these by dynamically running them in an interactive sandbox environment which helped us understand the common packing tactics.

Obfuscation vs. Packing. To differentiate between packing and other forms of anti-reversing in server-side web malware, we consider packing only when the original server-side code is already present in the packed file but is *not* present in its ready-to-execute form (i.e., it is encrypted, compressed, or otherwise transformed). In other words, the instructions to execute are later recovered and executed at runtime. Conversely, obfuscation is when the server-side malware is ready-to- execute, but is hard to understand for humans and/or automated tools. Mantovani et. al. [99] used a similar definition for obfuscation and packing in Windows binary executables. It is important to note that attackers could use both obfuscation and packing in the same code file.

4.1.1 Types of Obfuscation

Prior work [100] for web client-side obfuscation proposed a taxonomy for obfuscation categories. While the high-level definitions apply to server-side web malware, we wanted to confirm if attackers indeed used similar obfuscation techniques while considering the unique aspects of server-side code and data manipulation techniques. We manually analyzed the 826 files and confirmed 4 types of obfuscation.

1. Randomized Obfuscation. Attackers either (1) randomly insert instructions that preserve the code semantics or (2) repackage the malicious code for each victim with a randomized comment string within the code blocks, thus making the anti-malware solutions based on static signatures useless.

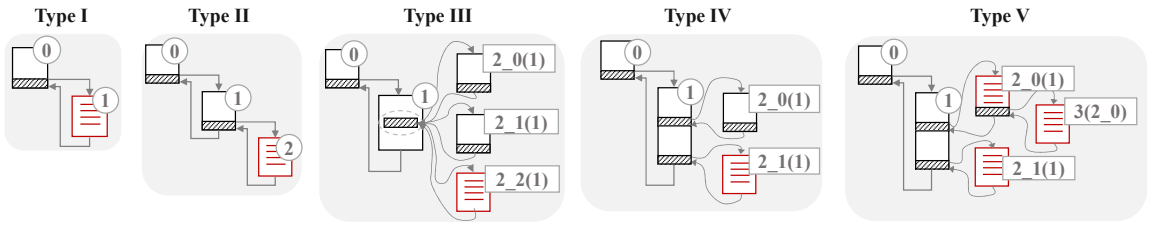

Figure 4.1: Types of Packing in Server-Side Web Malware.

2. Data Obfuscation. To hinder automated analysis, attackers use string manipulation tactics unique to server-side code. One common tactic is to obfuscate function and variable names by mapping ASCII characters to an alternate list of characters, and then reconstruct them. This increases the overhead of deciphering the underlying PHP code.

3. Encoding Obfuscation. Attackers use common encoders such as `base64_encode`, `urlencode`, or custom encoders to hide the malicious payload making it difficult to decipher. Furthermore, AVs cannot infer malicious code blocks that use encoders because these encoders can also be used within legitimate code.

4. Environmental Obfuscation. Attackers rely on PHP environment-specific interactions to hide inputs required to unpack or decrypt the malware payload, thus thwarting code readability. For example, malware authors often abuse PHP-specific semantics such as global variable definitions to obfuscate malware.

Based on these definitions, we found that the 826 obfuscated files contained 382, 551, 435, 117 files with randomized, data, encoding, and environmental obfuscation, respectively.

4.1.2 Types of Packing

Attackers use packing (i.e., transforming an executable code block and combining it with transformation logic into a single code file) to evade defenses. Listing 4.1 shows the pseudo-code for the most simple form of packed malware. Typically, packed malware contains an encoded or encrypted malware payload and decoding logic to decode the malware payload. Since the payload is decoded during execution, it resides

in memory and easily evades static analysis.

```
begin
    E = [encoded malware payload]
    K = extract_key()
    M = decode_malware_payload(E, K)
    execute(M)
end
```

Listing 4.1: Simple Packed Malware

We analyzed the 826 obfuscated files in a sandbox environment to identify packing. We wanted to study the complexities and in turn, gain an actionable understanding towards improving defenses against packed malware. We noted that 818 of these were packed and 8 were obfuscated without packing. Ugarte et. al. [101] provided a taxonomy of Windows malware packers based on the levels of complexity in the reconstruction of the target payload. This study assumed packing in the context of compiled executables. We translated the packing types to interpreted languages used in server-side web malware by considering the dynamic code generation capability unique to server-side code. To illustrate, Figure 4.1 shows the packing types seen in server-side web malware. The black rectangles represent the encoded, encrypted, or transformed code, the hatched rectangles represent the unpacking routine via dynamic code generation, and the red rectangles represent the unpacked malware payload.

1. Type I/Single Layer. This is the simplest case where a single unpacking or single dynamic code generation routine is executed before transferring the control to the unpacked malware payload program which resides in the first layer.

2. Type II/Multi-layer Linear. The unpacking routine is repeated multiple times sequentially to extract the malware payload. For example, layer 0 has an unpacking routine that extracts layer 1, which in turn generates the malware payload in layer 2.

3. Type III/Multi-layer Cyclic Tail Transition. In cyclic unpacking, there are backward transitions between the a layer and its predecessor. This is similar to the multi-layer linear unpacking, except the unpacking routine is executed in a loop.

4. Type IV/Multi-layer Cyclic Interleaved Single Frame. The unpacking routine has multiple backward transitions from several deeper layers that were generated from the same predecessor layer. Malware frequently use these interleaved transitions to set several intermediate variables necessary to unpack the final malware payload.

5. Type V/Multi-layer Cyclic Interleaved Multi- Frame Incremental. This is similar to Type IV, except the malware payload is unpacked incrementally. From all the Type V cases in our preliminary study, we found that malware incrementally unpacked function definitions for use in the subsequent layers.

By applying these definitions to our preliminary study dataset of 818 packed files, we found 47, 181, 33, 371, and 186 files with packing types I-V, respectively. Besides, when OBIWAN is applied on a temporally distributed dataset, it will reveal attack tactic evolution. Thus, it is both urgent and important to measure the obfuscation and packing types temporally and at scale.

4.2 Methodology

To understand the temporal evolution of web-based malware obfuscation over the last decade, we present OBIWAN, an automated obfuscation detection and unpacking framework. As shown in Figure 4.2, OBIWAN takes the webserver backup snapshots as input and analyzes the code files on every single snapshot. It first identifies the presence of obfuscation, and categorizes them into the 4 classes of obfuscation types (discussed in §4.1.1) by applying static analysis techniques. OBIWAN then dynamically analyzes these obfuscated files to identify instances of packing. It also collects the unpacked code layers and collects telemetry on each of

these layers such as the number of layers, the type of packing, hashes for each of the code layers, etc. OBIWAN then temporally correlates all of the collected results and metadata to extract web malware evolution patterns.

4.2.1 Obfuscation Detection And Categorization

We surveyed previous research on obfuscated web malware [42, 91]. While none of the previous works focused solely on studying obfuscation, we replicated their techniques to detect obfuscated code files in our dataset. In particular, OBIWAN identifies the presence of meaningless variables and functions, several code statements residing on a single line, encoding functions, code generation functions, string manipulation functions, multiple PHP blocks within a single code file, etc., for all code files in the nightly backups. With obfuscated code files identified for each snapshot, OBIWAN categorizes them and labels them with the obfuscation type used.

1. Randomized Obfuscation. OBIWAN generates the abstract syntax tree (AST) of the obfuscated code file, and tracks the code statements to find variables that are defined but not used within the code file. OBIWAN also scans for randomized comment strings inserted within the code files that change the code hash for each distributed variant. OBIWAN eliminates the false alarms from developers who forget to remove unused variables by ensuring that both randomization techniques are found in a given obfuscated file as confirmed by our preliminary study.

2. Data Obfuscation. OBIWAN parses the AST to identify the data manipulation tactics such as (1) array mapping, i.e., remapping ASCII characters to an alternate list of characters which are then used to define the encoded payloads; (2) the use of unicode characters to define payloads, function, and variable names; (3) string manipulation tactics such as splitting the suspicious function names to a list of discrete strings, and concatenating them together at runtime.

3. Encoding Obfuscation. Here, OBIWAN identifies all encoders used within the code file, including library encoders such as `base64_encode`, `urlencode`, etc.,

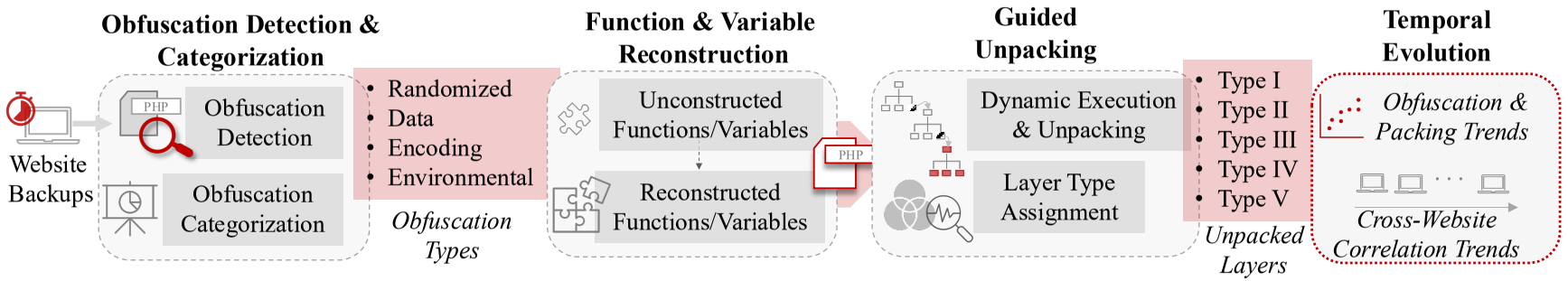

Figure 4.2: OBIWAN Pipeline for Obfuscation Detection and Unpacking.

and custom encoders using `xor` operations. If OBIWAN finds string concatenation operations within the AST, it reconstructions them (§4.2.2) and re-scans for all encoders within the code file.

4. Environmental Obfuscation. OBIWAN analyzes the AST to first identify all inputs to the obfuscated code file. If any of these inputs rely on PHP environment-specific interactions such as inputs from superglobal variables, or inputs from within the webserver filesystem, OBIWAN labels it as environment obfuscation.

Obfuscation is used both by web developers to prevent intellectual property theft and by attackers to hide their malicious payloads. This research focuses on malware obfuscation. Therefore, OBIWAN eliminates the benign files that are obfuscated for code protection. One lazy solution to eliminate benign obfuscation from malicious obfuscation is to identify commercial obfuscator signatures (e.g., Zend Guard encoder and obfuscator [102], ionCube encoder [103]). This is because benign code files typically rely on commercial solutions for obfuscation and attackers used custom obfuscation techniques. However, attackers have also evolved to use commercial obfuscators, thus making the commercial obfuscator detection an unreliable indicator.

To address this challenge, we analyzed the obfuscated files from a subset of our dataset to identify commonly used benign obfuscation patterns that are different from obfuscated malware. We found that, invariably, all benign obfuscation was employed by websites built on content management systems, and they either used (1) encoding obfuscation alone with well-known library functions such as `base64_encode` or (2) commercial obfuscators. Also, malicious Based on this observation, OBIWAN eliminated the benign obfuscated files from the malicious ones by identifying if (1) it used a single obfuscation type, typically encoding obfuscation *and* (2) the obfuscated code included commercial obfuscator signatures or headers.

4.2.2 Function & Variable Reconstruction

A common challenge encountered across malicious obfuscated files is that they often take advantage of the dynamic nature of PHP to construct function and variable names during execution. This ensures that the visibly suspicious activities such as downloading malicious content from an external server, or exfiltrating data from the webserver remain hidden to the website owner or AVs who employ static analysis techniques at best to analyze web malware. Often times the static analysis techniques look for code patterns [42] or specifically named dataflow sinks [84] to uncover malicious behavior.

To address this challenge, when OBIWAN parses at the AST and finds instances of functions and variables that are resolved at runtime, and statically performs function and variable reconstruction as shown in Figure 4.2. In particular, it iterates over the AST to replace the dynamically constructed functions and variables. First, it finds all assignment statements that create a variable or a function name which are later referenced in the obfuscated code file. It then traverses the AST and evaluates the right-hand-side of these assignments, and replaces the corresponding AST nodes with the computed value. OBIWAN replaces each function or variable name in the order that it was encountered it in the original AST, thus incrementally modifying the AST during traversal. Note that OBIWAN accounts for assignments to the same variable on different lines by using the json-encoded AST, which also encapsulates the line number information. This also replaces all future assignments of these functions or variables with its reconstructed value.

While evaluating the right-hand-side of assignments can be a risky technique, as it executes code contained in a malicious file without first understanding the behavior of the code, the vast majority of assignment statements contain simple string concatenation or other benign operations that do not pose a risk to the

execution environment. To mitigate the risks of disrupting the webserver, OBIWAN ensures that the code statements that are being computed only perform data manipulation operations. Also, OBIWAN performs all analyses in a sandboxed environment that contains a bare bones PHP interpreter with no internet connection. We found that this technique is effective at revealing the behavior of otherwise difficult to analyze malware, thus the benefits of executing small portions of potentially malicious code outweighs the risks of transforming otherwise impossible to classify malware into easily detectable programs.

4.2.3 Guided Unpacking

After resolving all of the functions and variables in the obfuscated code file, OBIWAN then identifies the presence of packing by dynamically executing it in a containerized environment. In particular, OBIWAN sends the obfuscated (and potentially packed) file to the containerized environment, which in turn returns the nested layers extracted during the unpacking process. The top layer represents the packed file, and the deepest layer represents the fully unpacked file. If the dynamic execution does not return any additional layers, it could indicate 2 possibilities: (1) the obfuscated file only uses obfuscation techniques that thwart the readability of the file and does not generate dynamic code, or (2) OBIWAN does not have access to the inputs necessary to navigate to the conditional branches that generate nested layers. OBIWAN records these as packed for further inspection.

OBIWAN's dynamic execution environment contains a modified PHP interpreter that hooks into the dynamic code execution functions (e.g., `eval`, `create_function`, etc.) to collect code layers that are dynamically generated by the packed file. The interpreter executes the code file, and when it finds a dynamic code execution function, it collects the code input to this function and records it as a code layer. Note that OBIWAN also verifies that the data recorded as a code layer are indeed additional code blocks by generating its AST.

For example, Listing 4.2 shows a simple packed webshell. When OBIWAN executes this, it hooks at the dynamic code execution function, i.e., `eval()`, and collects the input to this function. In this case, the base64 decoded result shown in Listing 4.3 is the input to `eval()`. The original packed code is labeled as `Layer 0`, and the unpacked code is labeled as `Layer 1`. While this was a simple example, OBIWAN can successfully unpack both all 5 types of packing listed in §4.1.2. Since the entire context of the website being analyzed is mounted into the containerized environment, OBIWAN can also resolve inter-file dependencies as well as inter-layer dependencies.

```
<?php
eval(base64_decode("PD9waHAKICAgIGlmKGlzc2V0
  KCRfR0VUWydjbWQnXSkpCiAgICB7CiAgICAgICAgc
  3lzdGVtKCRfR0VUWydjbWQnXSk7CiAgICB9Cj8+"));
?>
```

Listing 4.2: Simple Packed Webshell

```
<?php
    if(isset($_GET["cmd"]))
    {
        system($_GET["cmd"]);
    }
?>
```

Listing 4.3: Simple Unpacked Webshell

Packing Type Assignment. We identify the original packed malware as the 0^{th} layer in its packing tree, or its layer ID (L_ID) is 0. To assign packing types, for every

generated layer, OBIWAN records if (1) the transitions between them are linear or cyclic, (2) if the loops generated the layers, and (3) if a layer is a malware payload or an intermediate layer. Based on these measurements, it assigns the packing types as described in §4.1.2. To distinguish between intermediate layers and malware payloads, OBIWAN checks all of the functions used in each layer. If the functions in a layer *only* perform the unpacking routine (i.e., dynamic code generation and decryption or decoding), then OBIWAN assigns it as an intermediate layer. Functions that perform all other activities are marked as the malware payload. Note that in Type V packing, both malicious functions and unpacking routines can co-occur.

For linear transitions between layers, the layers are identified based on its depth, i.e., its L_ID = L_ID_{prev} +1. However, when there are cyclic transitions, the L_ID always encapsulates the predecessor layer that generated it in parenthesis. If a layer generates multiple layers, the L_ID also encapsulates the number that records this multiplicity. Figure 4.1 shows the layer IDs assigned for the cyclic packing cases (i.e., Types III, IV, and V). For the Type V example shown in Figure 4.1, L_ID 1 has two unpacking routines which generates layers 2_0(1) and 2_1(1). This means that, these 2 unpacked layers are at depth 2, multiplicity 0 and 1, respectively, and were extracted from the predecessor layer 1. However, at layer depth of 3, there is only a code frame extracted, and hence the multiplicity is dropped.

One challenge that became apparent while designing the unpacker is the inflation in the number of packed files, i.e., it could produce packed layers from files that are dependencies of the original packed file, which could lead to (1) flagging benign code files with malicious packed file dependencies as malicious, and (2) double counting the packed layers across multiple files. To address this, OBIWAN records the source filename and enumerates all dynamic code executor invocations, which allows it to differentiate the invocations from the top-level packed files and those from its dependencies during the course of the execution. In addition, OBIWAN also has a

timeout mechanism designed carefully to optimize for performance.

```php
<?php /*ynlpnweob*/
if (!defined('ALREADY_RUN_1bc2...'))
{
define('ALREADY_RUN_1bc2...', 1);

$dhezhdfhb = 3784;
function wmcjryjhyl($rrnulyr, $adesgv){$pvagpw = ''; for($i=0; $i < strlen($rrnulyr); $i++){$pvagpw .= isset($adesgv[$rrnulyr[$i]]) ? $adesgv[$rrnulyr[$i]] : $rrnulyr[$i];}
$gbnkuskbz="base" . "64_decode";return $gbnkuskbz($pvagpw);}
$ofotqmhu = 'e6rSCVsqXDnhbzQ5YCcnboZHbnn'.
...
...
$qphnzynoi = Array('1'=>'h', '0'=>'w', '3'=>'F', '2'=>'t', '5'=>'0', '4'=>'L', ..., 'x'=>'5', 'z'=>'2');
eval/*ynlpnweob*/(wmcjryjhyl($ofotqmhu, $qphnzynoi));
}
?>
```

Listing 4.4: Obfuscated and Packed Code Snippet

Listing 4.4 shows a code snippet for an obfuscated packed file. In particular, I present this to highlight the semantically irrelevant features used by attackers. For example, the variable `$dhezhdfhb` has been defined but is unused. Similarly, attackers also inject random comments. In Line 1 and Line 13, attackers used the same random string (`/*ynlpnweob*/`) in a comment which helps them easily evade hash-based detection systems.

4.2.4 Temporal Evolution

OBIWAN identified obfuscated code files, categorized them, and unpacked them to capture the code layers. Since OBIWAN performed these analyses on all server-side backup files from the nightly snapshots dating back to 2012, this gives us the unique vantage point to study how obfuscation and unpacking techniques have evolved over time. In particular, OBIWAN summarizes the overall trends in the types of obfuscation used to inform the existing defenses. It also captures metadata for all obfuscated files and its unpacked code layers. In particular, it records the file hashes as well as the normalized file hashes (after removing randomization obfuscation) and cross-correlates across websites to understand the prevalence of any large-scale malware campaigns and their persistence. OBIWAN also compares how packing techniques have evolved over time. It performs a cross-website temporal correlation to identify if any of the previously known malware payloads are repackaged and redistributed over time. This helps us assess the state of the current defenses and provide recommendations to strengthen them.

4.3 Evaluating OBIWAN

Before deploying OBIWAN at scale, we evaluate its design considerations. We chose 65 unique websites with packed files from our preliminary study. To this set, we added 35 randomly selected unbiased websites from our collaborator's dataset, giving us a total of 100 websites' nightly backups collected between April 2018 and March 2022 (a total of 12,554 nightly snapshots). We use these 100 websites as our evaluation dataset to establish ground truth. For this evaluation, we used a local workstation running Ubuntu 16.04 with 32GB memory and 8 x 3.60GHz Intel Core i7 CPUs.

4.3.1 Obfuscation Evaluation

Ground Truth. We first evaluate OBIWAN's obfuscation categorization. To do this, we use OBIWAN to identify and categorize obfuscated files in our dataset of

Total #Websites: 100 (65 bad + 35 unbiased)
Total #Files: 1,537,948
Total #Obfuscated Files: 7,849

Obfuscation Type	#W	#Obf. Files				
		GT	OBW	TP	FP	FN
Randomization	39	3,635	3,638	3,635	3	0
Data	43	5,260	5,299	5,260	39	0
Encoding	44	4,094	4,109	4,094	15	0
Environment	38	1,144	1,212	1,144	68	0
Total[1]	67	7,849	7,849	-	-	-

1:This is not the sum of the columns, but the total #websites and #files with code obfuscation.

Table 4.1: Evaluation of OBIWAN's Obfuscation Categorization

100 websites' nightly backup snapshots. We evaluate if it correctly categorizes the obfuscated files per the description in §4.2.1. As shown in Table 4.1, of the 1,537,948 files across 100 websites, OBIWAN identified 7,849 obfuscated files. Our team manually analyzed these files and tagged them with the corresponding obfuscation types for each file to obtain the ground truth. Note that any obfuscated file can belong to more than one category. To the best of our knowledge, OBIWAN did not produce any FNs as our team skimmed the remaining code files and could not find any obfuscated file OBIWAN missed.

Detection Results. In our dataset of 7,849 obfuscated files across 100 websites, OBIWAN reported a total of 67 websites (#W) with obfuscated files. Table 4.1 drills down into the different obfuscation types in this dataset. Columns GT and OBW show the number of ground truth files and those detected by OBIWAN, respectively, for each obfuscation category. Columns TP, FP, and FN present the corresponding true positive, false positive, and false negative numbers. OBIWAN labeled 3,638 files as randomized obfuscation with 3 FPs (Row 1). All 3 FPs belonged to the same website. After further analysis, we determined that the flagged randomized string was not a typical random string used for hash change; instead, it appeared to be a typo in

Initial Dataset: GT Total #Packed Files: 892 Initial Dataset TP: 868						Initial Dataset FN: 24				
	Packed Files					Packed Layers				
	GT	OBW	TP	FP	FN	GT	OBW	TP	FP	FN
Type I	51	48	48	0	3	51	48	48	0	3
Type II	194	193	193	0	1	736	733	733	0	3
Type III	36	33	33	0	3	258	247	247	0	11
Type IV	392	395	392	3	0	1,341	1,352	1,341	11	0
Type V	199	199	199	0	0	763	763	763	0	0
Total	872	868	865	3	7	3,149	3,143	3,132	11	17

Table 4.2: Evaluation of OBIWAN's Unpacking Module.

one of the comments. Data and encoding obfuscation were most commonly used by attackers. OBIWAN labelled 5,299 and 4,109 files as data and encoding obfuscation with 39 and 15 FPs, respectively (Row 2 and Row 3). It also categorized 1,212 files as environmental obfuscation, with 68 FPs (Row 4). For all of these cases, the FPs originated from the data manipulation found in benign code to which attackers appended obfuscated code blocks as part of the compromise.

4.3.2 Unpacker Evaluation

Dataset. Of the 7,849 obfuscated files, we eliminated 1,012 files that were not packed, i.e., they thwart code readability without generating new code dynamically. To identify duplicate files in the remaining 6,837 file dataset, we uniformly reformatted each file by removing semantically irrelevant features such as comments and unused variables. This resulted in our unpacker evaluation dataset of 892 unique packed files.

Ground Truth. When a file is packed, OBIWAN can either (1) unpack it entirely and generate all the hidden code layers, (2) partially unpack it, or (3) cannot unpack it. To generate the ground truth for OBIWAN's unpacker, we first used it to generate all of the unpacked layers. We manually investigated these layers, particularly by looking for the presence of any dynamic code generation instances that were not executed within each layer. During the unpacking process, if

OBIWAN did not encounter *any* dynamic code generation instance, we can safely conclude that it did not unpack the input file. When a dynamic code generation instance was not executed in any of the intermediate layers, it is likely that OBIWAN only partially unpacked it. For all the files that were either partially unpacked or not unpacked, we tried to manually unpack them by forcing execution of the unreachable code blocks in a sandbox. Based on this, we derived the ground truth for OBIWAN's unpacker by tagging each of the original packed files with its corresponding packing type (introduced in §4.1), and the number of expected layers it should generate. We ran OBIWAN's unpacker on all of these files and compared the packing types and the number of unpacked layers with the manually derived labels.

Detection Results. Table 4.2 presents the unpacker evaluation results. In our initial dataset of 892 (GT) packed files, OBIWAN successfully unpacked 868 files (TP). Since OBIWAN did not fully unpack 24 files, we mark these as FNs. Of the 24 files, it partially unpacked 16 files and failed to unpack 8 files. OBIWAN could not unpack these files because they contained dynamic code generation instances that were not in the execution path and relied on attacker input configuration. However, we found that input-dependent unpacking behavior was uncommon and seen in about 2% of the ground truth packed files.

The first half of Table 4.2 details the number of packed files corresponding to packing complexity Types I-V. Of the 892 packed files, we could derive the packing types ground truth for only 872 files. We could not assign packing types for 20 of the 24 files that relied on attacker inputs for successful unpacking. We manually unpacked 4 of the partially unpacked files by forcing the execution along the path validated by attacker input and found that 3 of these used Type I packing (Row 1 Column 5) and 1 used Type II packing (Row 2 Column 5). We found that OBIWAN correctly assigned the packing types for 865 of the 868 files (Total Row, Column

3) that it successfully unpacked. It incorrectly categorized 3 Type III files (Row 3 Column 5) as Type IV (Row 4 Column 4). This is because these Type III files first unpacked a code layer and then unpacked the malicious payload within a loop of a single iteration.

The second half of Table 4.2 shows the layers generated by unpacking each of the packing types. OBIWAN extracted a total of 3,143 layers (Total OBW Packed Layers). Since OBIWAN missed identifying 4 packed files (3 Type I FN and 1 Type II FN), it also missed the layers these packed files generated. We manually unpacked them and found that these files generated a total of 3 Type I layers (Row 1 Column 10) and 3 Type II layers (Row 2 Column 10). Note that OBIWAN correctly extracted all of the 11 layers from the 3 Type III packed files it originally misclassified as Type IV (Row 3 Column 9 and Row 4 Column 10). From Table 4.2, the overall accuracy of labeling the packing types is 99.1%. Thus we conclude that, OBIWAN can accurately unpack and label the packed files.

4.4 Temporal Evolution of Packed Files

Total #Packed Files: 892				
Year	#Packed Files	#Layers	#L/PF	#Reused Layers
2018	265	609	2.3	-
2019	198	594	3.0	129
2020	179	752	4.2	153
2021	138	678	4.9	121
2022	88	510	5.8	92
Total	868	3,143	3.6	495

Table 4.3: Temporally Evaluating Unpacked Files.

Table 4.3 shows all the temporal distribution of the 868 files unpacked by YODA between 2018 and 2022. In particular, it shows the number of packed files and the number of layers generated from these packed files corresponding to the year they

Total #Packed Files: 892			
Unpacker	#Unp. Files	Time/File	#Timeouts
OBIWAN	868	0.2 s	18
UnPHP	126	128 s	766

Table 4.4: Comparing OBIWAN's Unpacker with UnPHP.

first appeared in our dataset. Column #Reused Layers shows the number of layers that appear in a given year and were previously seen in previous years in this dataset. As seen in Table 4.3, the number of reused layers is never more than the number of packed layers. We investigated the reused files and found that attackers only reuse the deepest layers containing the malware payload, and using different encoding and packing methods to evade detection. Also, from the Column #L/PF (the average number of layers generated by the packed files), it is evident that even though the malware payloads are reused, the number of packed layers has increased over time from 2.3 in 2018 to 5.6 in 2022.

4.4.1 Comparing With UnPHP

Since many research solutions to unpack server-side web malware are not publicly available, the research community has relied on commercial solutions such as UnPHP [93]. We used OBIWAN and UnPHP to unpack the unique, packed files in our evaluation dataset and compared their performance (Table 4.4). While OBIWAN could successfully unpack 868 of the 892 packed files in our dataset, UnPHP could only unpack 126 files. UnPHP's API takes 128s on average to unpack a file, whereas OBIWAN takes 0.2s. We assigned a 30 minute timeout per file for both unpackers and found that OBIWAN exited with a timeout for only 18 files and exited without a timeout by partially unpacking 6 files. Conversely, UnPHP timed out for 742 files and did not produce any unpacked layers. Thus, we can confirm that OBIWAN's performance is superior to the widely accepted unpacker.

4.5 Large-Scale Study

#Websites:	501,003	Duration: 2012 - 2022	
Min. Duration	96 days	Min. #Files	0.4K
Avg. Duration	682 days	Avg. #Files	16.7K
Max. Duration	3,717 days	Max. #Files	397K

Table 4.5: Dataset Summary.

Dataset. Our dataset consists of 501,003 unique websever backups collected between 2012 and 2022, as summarized in Table 4.5. The backups contain an average of 682 day-snapshots per website. Many backups were collected since 2012, with a maximum of 3,717 day-snapshots per website. Each website has between 400 to 379K files, with an average of 16.7K files per website. I built upon `cmsscanner` [104] to detect the underlying CMS the website owners used. I improved it to support detection of additional popular CMSs on the market such as Square Space, Shopify, Big Commerce, and Woo Commerce. In particular, I scan the local filesystem of the server-side backup snapshots to identify the obvious indicators such as metadata headers and tags. When these are unavailable, I use the less obvious indicators such as the directory structure, code patterns, and non-open-source function calls to identify the CMS platform. This improved `cmsscanner` is available at: https://cyfi.ece.gatech.edu/.

Table 4.6 presents the number of websites (#W) using the CMSs in Column 1. Over 73% of the websites in our dataset are built on WooCommerce, WordPress, and Big- Commerce[1]. This follows the in-the-wild CMS trend; over 60% of CMS-based websites are built on WordPress [83]. We deploy OBIWAN on this dataset to perform a decade-long retroactive study of obfuscation and packing types in server-side web malware.

[1] WooCommerce and BigCommerce and e-commerce platforms built on the WordPress core.

CMS	#W	#OW	#OF	#F/W	Obfuscation Type				#PF	Packing Type					#L	#L/PF
					Rnd.	Dta.	Enc.	Env.		I	II	III	IV	V		
WooCommerce	160,441	8,022	2.2M	276	872K	1.2M	1.1M	481K	1.8M	105K	402K	72K	816K	417K	8.9M	4.9
WordPress	106,620	6,930	2.9M	425	1.2M	1.7M	1.4M	663K	2.6M	121K	582K	105K	1.2M	634K	15.2M	5.8
Big Commerce	100,563	4,238	2.0M	482	1.4M	1.9M	1.6M	753K	1.9M	93K	413K	74K	836K	442K	15M	8.1
Shopify	46,328	1,009	102K	101	41K	59K	49K	23K	73K	6.4K	16K	2.9K	33K	15K	168K	2.3
Magento	8,424	996	460K	462	188K	267K	222K	104K	368K	14K	82K	15K	166K	92K	1.5M	4.1
SquareSpace	6,685	133	31K	232	15K	21K	18K	8K	24K	1.4K	5.4K	973	11K	5.6K	114K	4.7
Joomla	2,780	338	24K	72	14K	20K	17K	8K	20K	1.3K	4.4K	788	9K	4.3K	71K	3.6
Drupal	2,056	308	78K	255	41K	58K	48K	23K	54K	2.5K	12K	2.2K	24K	13K	239K	4.4
Prestashop	1,952	97	65K	664	26K	37K	31K	14K	46K	2.3K	10K	1.8K	21K	11K	306K	6.7
PivotX	1,428	62	13K	213	5.4K	8K	6K	3K	10K	500	2.3K	417	4.7K	2.5K	45K	4.3
Concerte5	484	31	349	11	142	201	168	79	280	17	63	12	126	62	924	3.3
Nextcloud	61	4	572	143	233	331	276	129	487	12	109	20	220	126	2.8K	5.8
TYPO3 CMS	37	2	393	197	160	227	189	89	335	19	75	14	151	76	905	2.7
Matomo	24	1	20	20	8	11	9	5	17	0	6	0	8	3	140	8.2
Unknown	63,120	5,617	2.1M	371	1.3M	1.8M	1.5M	710K	1.8M	93.9K	393K	71K	797K	416K	12M	6.7
Total	501,003	27,788	10.1M	362	5.1M	7.2M	6.0M	2.8M	8.7M	441K	1.9M	346K	3.9M	2.1M	54M	6.2

Table 4.6: Obfuscation and Packing Landscape In Our Dataset.

4.5.1 Obfuscation & Packing Landscape

Table 4.6 presents the overall results after deploying OBIWAN on our dataset. OBIWAN found a total of 10.1M+ obfuscated files (#OF) across 27,788 websites (#OW). Even though WooCommerce has the highest number of obfuscated websites (8K), we find that WordPress has the highest number of obfuscated files in our dataset. Column 4 (#F/W) shows that Prestashop, BigCommerce, WordPress, and Magento have a high ratio of obfuscated files per obfuscated website. This means that the attackers have more control to introduce redundancy in these websites while dropping obfuscated files when compared to other CMSs.

Columns 5-8 in Table 4.6 show the Obfuscation Type distribution. Of the 10.1M obfuscated files, total of 5.1M, 7.2M, 6M, and 2.8M files contained randomized, data, encoding, and environmental obfuscation, respectively. When we analyzed individual obfuscated files, we found that attackers often used multiple obfuscation techniques in a single file; data obfuscation was commonly seen in tandem with randomized or encoding obfuscation. Regardless of the underlying CMS, most of the websites had a majority of files with data and encoding obfuscation. This shows that attackers heavily rely on data manipulation techniques to evade detection.

The second half of Table 4.6 shows the distribution of packed files in our dataset. Of the 10.1M obfuscated files, OBIWAN found a total of 8.7M packed files; 1.4M files were not packed and only used obfuscation techniques that did not generate any dynamic code. OBIWAN then categorized these packed files based on the packing types I-V (described in §4.1.2). The vast majority of the packed files either use multi-layer linear (Type II), multi-layer cyclic interleaved single-frame (Type IV), or multi-frame incremental (Type V) packing. The 8.7M packed files generated a total of 54M layers (#L) upon unpacking. Packed files in BigCommerce and Matomo CMSs had the highest average number of layers per packed file (#L/PF) of 8.1 and 8.2, respectively. This means that on average, every packed file generated 8 layers on

unpacking. While this is true on average, we found few packed files with a maximum of 48 layers per packed file.

Comparing with Windows Malware Packing. Ugarte *et al.* [101] showed the distribution of packing types in traditional Windows malware. Over 50% of the traditional Windows malware use Type III packing. In comparison, less than 4% of the server-side web malware (346K of 8.7M) use Type III packing. Due to the ease of dynamic code generation for interpreted languages, server-side web malware rely on the less frequently used packing types in traditional Windows malware. No more than 14% of the Windows malware use Type IV packing, and less than 1% use Type V packing. Conversely, these packing types are the norm for server-side web malware — 45% use Type IV (3.9M of 8.7M), and 24% use Type V packing (2.1M of 8.7M). We find that Type VI packing for Windows malware at a page, function, and basic block granularity does not translate to server-side web malware since all packing happens at a code block level. Only 12% of Windows malware use multi-layer linear Type II packing, whereas almost 22% of server-side web malware (1.9M of 8.7M) use Type II packing. We note that 25% of Windows malware use the simplest Type I packing. However, only a little over 5% of server-side web malware (441K of 8.7M) rely on Type I packing. While it may appear that the sever-side web malware use sophisticated packing techniques, it is in fact, the opposite. As we will see later (in §4.5.2), attackers leverage the dynamic nature of interpreted languages to achieve superior polymorphism using minimal effort.

4.5.2 Do Attackers Reuse Malware?

Next, we turned our attention to studying the temporal distribution of packed files to understand if attackers reuse packed malware. Table 4.7 presents the summary. Over the last decade, our dataset had a total of 8.7M packed files (#PF) that generated 41M intermediate packed layers (#IL) and 12.5M unpacked malware payloads (#UP). From this table, it is clear that attackers have consistently used packing over time.

Year	Packed Files					Intermediate Layers					Unpacked Payload				
	#PF	#RPF	%RPF	#RNPF	%RNPF	#IL	#RIL	%RIL	#RNIL	%RNIL	#UP	#RUP	%RUP	#RNUP	%RNUP
2012	13K	-	-	-	-	25.1K	-	-	-	-	15.7K	-	-	-	-
2013	78K	4.7K	6.0%	5.5K	7.0%	153K	34.1K	22.3%	34.3K	22.4%	82.6K	24.0K	29.0%	24.1K	29.2%
2014	134K	9.9K	7.4%	12.1K	9.0%	330K	73.2K	22.2%	73.2K	22.2%	162K	54.9K	33.8%	55.1K	33.9%
2015	208K	14.1K	6.8%	19.7K	9.5%	619K	127.2K	20.5%	128.3K	20.7%	258K	89.2K	34.6%	89.4K	34.7%
2016	235K	12.4K	5.3%	68.0K	29.0%	875K	410.7K	46.9%	411.0K	47.0%	279K	113K	40.5%	115K	41.1%
2017	547K	43.2K	7.9%	127K	23.3%	2.1M	774.2K	36.1%	775.4K	36.1%	635K	246K	38.7%	247K	39.0%
2018	782K	71.1K	9.1%	251K	32.1%	3.2M	1.52M	48.3%	1.53M	48.4%	969K	412K	42.5%	414K	42.7%
2019	800K	49.6K	6.2%	289K	36.1%	3.6M	1.76M	48.6%	1.77M	48.7%	908K	421K	46.4%	424K	46.8%
2020	3M	105K	3.5%	720K	24.0%	14M	4.34M	31.0%	4.34M	31.0%	5M	3.19M	63.4%	3.21M	63.7%
2021	2.4M	118K	4.9%	463K	19.3%	13.3M	2.79M	21.1%	2.8M	21.1%	3.6M	2.49M	70.3%	2.5M	70.5%
2022	467K	13.1K	2.8%	98.0K	21.0%	2.7M	600.2K	22.0%	601.8K	22.1%	602K	417K	69.3%	419K	69.7%
Total	8.7M	441K	5.1%	2.05M	23.7%	41M	12.44M	30.4%	12.47M	30.4%	12.5M	7.46M	59.7%	7.5M	60.0%

Table 4.7: Temporal Evolution For Packing Based On Malware Reuse. Here, PF - Packed Files; R - Reused; RN - Reused and Normalized; IL - Intermediate Packed Layers; UP - Unpacked Payload

However, the use of packing has surged since 2020. About 3M packed files were introduced in our dataset in 2020 alone.

The columns #RPF, #RIL, and #RUP show the number of reused packed files, intermediate layers, and unpacked malware payloads, i.e., the number of files in any given year that appeared in our dataset in any of the previous years. We measure this by checking if the file hash was previously seen. For example, in 2014, of the 134K packed files that were introduced in our dataset, 9.9K files (Row 3 Column 2) were previously seen across other websites in 2012 and 2013; or 7.4% of the packed files (%RPF) were reused in 2014. Similarly, these 134K packed files in 2014 generated 330K intermediate layers and 162K unpacked malware payloads; 22.2% (73.2K in Row 3 Column 7) of the intermediate layers and 33.8% (54.9K in Row 3 Column 12) of the unpacked malware payloads were reused.

Overall, we see that only 5.1% of the total #PF were reused as-is. This could be attributed to the same attacker targeting multiple websites or attackers adding redundancy by dropping several duplicate packed files within the compromised website. From Table 4.7, we also see that attackers reused 30.4% of total #IL and over 59% of the total #UP. During 2013-2019, even though attackers repacked pre-existing malware payloads (i.e., #RUP), we found that since 2020, at least over 60% of the malware payloads were recycled by packing them in multiple layers, and over 70% of the malware payloads seen in our dataset in 2021 were reused. This goes to show that the attackers are not reinventing the wheel but merely using simple multi-layer packing techniques to evade detection.

Since attackers use randomization tactics (e.g., random comments, redundant variables, etc.), we normalized each layer by removing the redundancies and uniformly reformatted them. We also recorded the hashes for each of the normalized layers and compared them temporally to study their prevalence. The column names with an 'RN' prefix refer to reused and normalized. As seen in Table 4.7, the reused

top-level packed files increased from 5.1% to 23.7% (total %RNPF) upon normalization. This highlights that attackers used randomization tactics on pre-existing packed malware samples before distributing them. When these samples are unpacked, they will generate the exact same intermediate layers and the malware payloads. This explains the reason for higher reuse in intermediate layers (30.4% %RIL) despite low reuse in packed files (5.1% %RPF). We also noted a spike in the use of randomization tactics in the top-level packed file in 2016 from 9.5% (Row 4 Column 5) to 29% (Row 5 Column 5), and these techniques have been used consistently since. We found that the increase in the total number of reused intermediate layers (12.44M to 12.47M) and unpacked payloads (7.46M to 7.5M) upon normalization (i.e., #RNIL and #RNUP) is marginal. We do not see these tactics being widely applied to the intermediate layers or the malware payload. This shows that attackers have relied on exerting bare-minimum effort to evade existing defenses by only modifying the top-level packed files.

4.5.3 Can AVs Detect Packed Malware?

We now analyze the efficacy of AVs in detecting server-side packed malware. We queried all packed files (#PF), the intermediate layers (#IL), and the unpacked malware payloads(#UP) on VirusTotal [98]. If 3 or more AV engines within VirusTotal report a file as malicious, we assign the file as detected malicious by AVs. Note that these files appeared in our dataset in the years presented in Column 1. However, they were queried against VirusTotal in April 2022. Table 4.8 shows a temporal distribution of AV evasion capabilities for the packed files in our dataset. The columns $\#PF_a$, $\#IL_a$, and $\#UP_a$ show the number of packed files, intermediate layers, and unpacked malware payloads that were identified as malicious by AVs. For example, of the 134K packed files, 330K intermediate layers, and 162.4K unpacked malware payloads in our dataset in 2014 60.8K packed files (or 45.3%), 7.6K intermediate layers (2.3%), and 116.5K unpacked malware payloads (71.7%)

Year	Packed Files					Intermediate Layers					Unpacked Payload				
	#PF	#PF$_a$	%PF$_a$	#NPF$_a$	%NPF$_a$	#IL	#IL$_a$	%IL$_a$	#NIL$_a$	%NIL$_a$	#UP	#UP$_a$	%UP$_a$	#NUP$_a$	%NUP$_a$
2012	13K	10.2K	78.3%	0.3K	2.3%	25.1K	0.3K	1.2%	6	0.02%	15.7K	12.8K	81.3%	2.8K	18.0%
2013	78K	63.8K	81.8%	2.6K	3.3%	153K	6.0K	3.9%	306	0.20%	82.6K	65.6K	79.4%	6.7K	8.1%
2014	134K	60.8K	45.3%	11.3K	8.4%	330K	7.6K	2.3%	727	0.22%	162.4K	116.5K	71.7%	3.4K	2.1%
2015	208K	82.3K	39.6%	26.8K	12.9%	619K	8.7K	1.4%	619	0.10%	257.8K	190.5K	73.9%	46.1K	17.9%
2016	235K	29.8K	12.7%	37.8K	16.1%	875K	25.4K	2.9%	595	0.07%	279.1K	191.7K	68.7%	26.0K	9.3%
2017	547K	144K	26.4%	64.6K	11.8%	2.1M	105.1K	4.9%	1480	0.07%	634.7K	389.1K	61.3%	62.5K	9.9%
2018	782K	232K	29.7%	106.3K	13.6%	3.2M	88.6K	2.8%	1012	0.03%	969.3K	663.0K	68.4%	188.0K	19.4%
2019	800K	119K	14.9%	145.6K	18.2%	3.6M	112.6K	3.1%	2868	0.08%	907.7K	527.4K	58.1%	60.8K	6.7%
2020	3.0M	126K	4.2%	260.9K	8.7%	14006.0K	378.2K	2.7%	756	0.01%	5038.5K	3164.2K	62.8%	433.3K	8.6%
2021	2.4M	93.6K	3.9%	200.3K	8.4%	13292.2K	412.1K	3.1%	1183	0.01%	3.6M	2194.5K	61.8%	277.3K	7.8%
2022	467K	28.5K	6.1%	31.3K	6.7%	2724.5K	49.0K	1.8%	242	0.01%	601.8K	384.6K	63.9%	70.8K	11.8%
Total	8.7M	991K	11.4%	888K	10.2%	40964.9K	1.2M	3.1%	9795	0.02%	12500.7K	7899.8K	63.2%	1177.8K	9.4%

Table 4.8: Packing Evolution Based On AV Evasion. Here, PF - Packed Files; N - Normalized ; IL - Intermediate Packed Layers; UP - Unpacked Payloads, Subscript$_a$ - identified as malicious by AVs

were identified as malicious by AVs.

Overall, 11.4% of all packed files in our dataset were identified as malicious by AVs. It is interesting to note that AVs' rate of malware detection has been decreasing over time. While over 78% of the top-layer packed files from 2012 and 2013 were identified as malicious, less than 6% of the packed files that appeared after 2020 were detected. AVs can accurately detect older packed malware, and attackers are largely being successful at evading AVs recently. As seen from the $\%IL_a$ column, an average of 3.1% of the intermediate layers were identified as malicious by AVs, i.e., they have consistently failed at detecting the intermediate layers generated by the packed malware. This is proof that the vast majority of AVs do not unpack packed server-side web malware and hence cannot detect the unpacked layers as malicious. As shown in the highlighted $\%UP_a$ column in Table 4.8, over 63% of unpacked malware payloads were identified as malicious. Since attackers constantly reuse malware payloads without modifying them, AVs can identify them as malicious. From these results, we can infer that AVs rely on pattern-based or hash-based detection to identify server-side web malware.

We also normalized the packed files and queried these against VirusTotal to study the effect of randomization on AV detection. We found that the AV detection rate dropped upon normalizing the files — Upon normalizing, (1) the top-layer packed file detection rate dropped from 11.4% to 10.2%, (2) the AV detection for the normalized intermediate layers dropped to a negligible 0.02%, and (3) the unpacked payload detection heavily plummeted from 63.2% to a mere 9.4%. Even though attackers rarely use randomization tactics at the malware payload level, a simple change by uniformly formatting the code makes it go unrecognized by most AVs. This further supports our hypothesis that AVs rely on hash-based or pattern-based detection systems. Overall, attackers successfully evade AVs by packing malware payloads. Even though AVs can detect malware payloads that are

not packed, the detection rate is low. There exists a gap with AV detection; a large percentage of packed malware payloads still remain detected. It is both urgent and important to close this gap since the majority of the less-technical users with a web presence rely on AVs to safeguard their websites.

4.6 Case Studies

We present a deep dive into some of the most frequently seen packed malware samples across all websites in our dataset. The file hashes corresponding to the files in Table 4.10 and Table 4.11 are listed in Table 4.9.

Hash ID	MD5 Hash
Top 5 Packed Malware Hashes:	
F1	d41d8cd98f00b204e9800998ecf8427e
F2	6ec256c8a7669df51b63aea2878825e2
F3	d82e04bf26874d49951afce9472cb88d
F4	96d369cdf26790ef3ca5b2de3166cccb
F5	1e583673c90528a5d02466e98fa08d42
Top 5 Unpacked Payload Hashes:	
F6	cb6ee491fcdea60465dd0d9c695b15a8
F7	ba7e32ea875b51476de74ffac5725dba
F8	97e976310f5478034e5f11f436f37c25
F9	af05dc268567816a26fdcd10450620a7
F10	3f60851c9f7e37c0d8817101d2212c68

Table 4.9: Hashes For The Top 5 Packed Malware and Top 5 Unpacked Payloads In Our Dataset

4.6.1 Popular Malware

Top 5 Files	#PF	#W	#L/PF	VT Detect	First Seen	Last Seen
F1	65,328	4,832	2	Not Detected	Mar 2016	Feb 2022
F2	17,434	4,534	2	Not Detected	Jan 2016	Jan 2022
F3	1,377	1,221	4	Not Detected	Dec 2018	Mar 2022
F4	1,347	1,168	5	Not Detected	Jul 2019	Apr 2022
F5	2,464	987	6	Not Detected	Mar 2019	Mar 2022

Table 4.10: Top-5 Popular Layer 0 Packed Files

Table 4.10 shows the distribution of the 5 most popular top-layer packed files (i.e., layer 0) in the dataset. Attackers are distributing the exact same packed file to multiple victims. From OBIWAN's results, we found 65,328 instances of the packed file F1 (#PF, Column 2) distributed across 4,832 unique victims (or websites #W, Column 3). This was the most frequently seen packed file in our dataset that generated 2 layers upon unpacking (#L/PF, Column 4). We note that the older packed files (i.e., early First Seen dates) generated fewer layers per packed file when compared with the newer files. For example, F1 and F2 were first seen in early 2016 and they generated only 2 layers each upon unpacking. However, F4 and F5 introduced in 2019 generated 5 and 6 unpacked layers, respectively. We queried these files on VirusTotal to see if any of them were detected by AVs. As shown in Column 5; none of the files were detected by the AV engines represented on VirustTotal. Since attackers can carry out their malicious activity by going undetected, these packed files are still being distributed. Even though they were first seen in our dataset in 2016, Column 7 shows that they are still being circulated to newer victims in 2022. The lack of defenses for server-side web malware has made it easier for attackers to exploit unsuspecting victims.

4.6.2 Packed Layer Evolution

Top 5 Files	#UP	#W	Min. #L	Max. #L	First Seen	Last Seen
F6	112,896	16	7	7	Feb 2022	Apr 2022
F7	88,143	4,711	3	12	Mar 2016	Mar 2022
F8	51,811	213	4	9	Nov 2021	Apr 2022
F9	18,022	1,496	3	5	Mar 2017	Feb 2022
F10	13,792	448	2	18	Apr 2017	Mar 2022

Table 4.11: Top-5 Popular Unpacked Payloads

Table 4.11 presents the distribution of the 5 most popular unpacked payload files in our dataset. Recall that the same payload can be generated by several top-layer packed files. In other words, attackers can use various levels of packing to distribute

the same payload. F6 is the most frequently seen unpacked payload in our dataset — we found over 122K instances of F6 across 16 websites. It was interesting to find that 5K-7K instances of this payload were seen in the 16 websites. Besides, the same payload was packed exactly 7 times and was coupled with randomized obfuscation on the top layer before being distributed to the victims. We note that while F6, F8, and F10 are distributed across fewer websites, F7 and F9 are distributed across a large number of websites (greater than 1.4K) thus dropping fewer packed files per website on average. Columns 4-5 in Table 4.11 show the minimum and the maximum number of layers used by attackers to pack the payload. F10 is an interesting case — it was packed with a minimum of 2 layers when it was first introduced in our dataset in 2017; in Mar 2022, the same payload was packed in 18 layers. The most widely distributed (i.e., max. #W) payload, F6, has been around since March 2016, and only 2 of these top 5 payloads are newer and have been circulated since 2021.

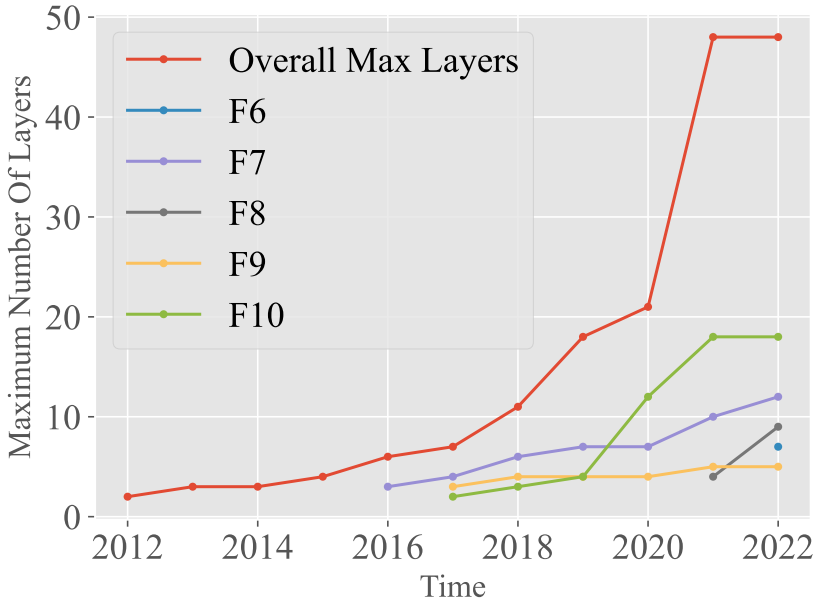

Figure 4.3: Maximum Packing Layers Seen Each Year

Figure 4.3 shows the temporal evolution of the maximum number of packing layers seen each year over the last decade for all malware payloads in our dataset (red line), and for the top-5 frequently seen malware payloads presented in Table 4.11. Overall,

attackers have been increasing the number of packing layers for malware payloads over time. In 2012, malware payloads were packed in 2 layers at most and has steadily increased to 21 layers in 2020. Over 300 malware payloads introduced in 2021 and 2022 were packed in 48 layers. From Figure 4.3 for the malware payloads F6-F10, it is clear that attackers are repacking the same malware payload over and over to evade detection.

CHAPTER 5
RELATED WORK

This section presents the literature review for techniques that enable advanced malware analysis of targeted web attacks, causality modeling via provenance inference and measurement studies.

5.1 Large-Scale Study of Web Attacks

Several studies have been published which used high-interaction honeypots to understand web attacks on a large scale [92, 105, 27]. In a large-scale web honeypot experiment, Canali and Balzarotti deployed 17 publicly accessible web shells that attackers could discover through the use of specially crafted search engine queries, also known as *dorks* [27]. Starov et. al. showed that most web shells are indeed backdoored but these backdoors are triggered as soon as the main page of the shell loads [92]. In a follow up study that investigated the role of web hosting providers in detecting compromised websites, Canali et al. [106], uploaded the arguably the most popular web shell, the c99 shell to 22 shared hosting providers and used simulated attackers to send commands towards the server. Only one of the 22 investigated shared hosting providers identified the malicious shell, even among the providers offering security services at an additional cost. Attacks against web applications have also been studied by using low-interaction honeypots [107, 108, 109].

Some techniques tried to assess the impact of web application compromises by studying the role of hosting providers [106] and understanding the response landscape by studying large-scale notification campaigns [110]. Similar to this research, Canali et.al. [27] also found attackers dropping large volumes of files on the web server. While these techniques focused on attacks targeting a generic web

application, this research studies the spread of multi-stage attacks on CMS-based websites and, more specifically, within real-world production websites. In particular, this study is designed to investigate such multi-stage attacks based on only the already collected nightly website backups.

Past research also studied web attacks as seen from the web browser [111, 112, 41, 113, 114, 115]. There is a large body of work studying social engineering and deceptive attacks [116, 117, 118]. In most of this work, the focus is on the technical mechanisms used by attackers to spread malware. The literature is presented by Clark et al. [119] provided the first analysis on survey scams by looking into Facebook spam URLs. Other studies used webserver backups [95] to understand web attacks on a large scale. Several techniques also studied the response landscape from post-compromise notification campaigns [110, 120]. While these studies focused on well-known web attack types, this research focuses on the spread of malware via CMSs.

Naderi et. al. [69] also analyzed nightly backups to investigate malware at an entire-website granularity. However their study is neither proactive nor fine-grained enough to vet previously-unseen code for malicious behavior. Their detection is coarse-grained as it relies upon strict temporal sequences of website-level indicators (e.g., stand-alone backdoor file injection followed by file deletion). Malicious plugins in CMSs do not exhibit overt temporal sequences of such indicators. They are deployed all at once and lie in wait until the website is loaded (e.g., blackhat SEO), requiring a CMS-centric detection and analysis.

5.2 Causality Modeling

Recent work suggests that data provenance might be a better data source for persistent threat detection. Data provenance represents system execution as a directed acyclic graph (DAG) that describes information flow between system subjects (e.g., processes) and objects (e.g., files andsockets). Historically, IDSs have

tended to produce alerts that are too numerous and low-level for human operators. Techniques needed to be developed to summarize these lowlevel alerts and greatly reduce their volume. There have been significant advances in identifying the provenance of an attack by monitoring the behavior of a system in order to reconstruct the chain of events that led to the attack [3, 121, 11, 122, 12, 6, 16].

Most works which focused on advanced multi-stage attack detection, e.g., HOLMES [2] and SLEUTH [15], are built on OS audit data and used system-call level logs for real-time analytics. A variety of security-related applications leverage provenance, mostly notably for forensic analysis and attack attribution [123]. BackTracker analyzes intrusions using a provenance graph to identify the entry point of the intrusion, while PriorTracker [16] optimizes this process and enables a forward tracking capability for timely attack causality analysis. HERCULE [13] analyzes intrusions by discovering attack communities embedded within provenance graphs. Winnower expedites system intrusion investigation through grammatical inference over provenance graphs, and simultaneously reduces storage and network overhead without compromising the quality of provenance data.

NoDoze [25] performs attack triage within provenance graphs to identify anomalous paths. Bates et al. [124] were the first to use provenance for data loss prevention, and Park et al. formalized the notion of provenance-based access control (PBAC). Ma et al. [11] designed a lightweight provenance tracing system ProTracer to mitigate the dependence explosion problem and reduce space and runtime overhead, facilitating practical provenance-based attack investigation. Pasquier et al. [125] introduced a generic framework, called CamQuery, that enables inline, realtime provenance analysis, demonstrating great potential for future provenance-based security applications.

However, these fine-grained log-based provenance tracking techniques require significant instrumentation and are hardly deployed by CMS hosting companies.

This research leverages what is already the industry standard (nightly backups) to model long-lived multi-stage attack progression via temporal correlation of spatial metrics and outlier detection.

5.3 Web Application Security

To preemptively secure websites against attack, recent research has focused on analyzing particular classes of attacks, such as ad injection [126, 127, 128], survey scams [129, 119], cross-site scripting [130, 131, 132, 133], PHP code injection [134, 135], SQL injection [136, 133, 137, 138], file inclusion attacks [139, 140], etc.

Other defenses against web application vulnerabilities include the following: (1) static analysis [141]; (2) dynamic analysis [142]; (3) combination of static and dynamic analysis [143]; (4) input validation [144]; and (5) fuzz testing [145], but they have well-known limitations. For instance, most static analysis tools have large false positives, and input validation methods are specific to certain web attacks such as SQL injection.

These research techniques focus on individual layers of web applications. However, since CMSs contain code across all of these layers and are marketed to less-technical website operators, attack-vector-specific solutions are not commonly deployed. This research is attack-vector agnostic and enables the investigation of a compromised CMS post-attack.

5.4 Web Malware Analysis

Recent web-based malware analysis research analyzed targeted attack classes like webshells [67, 68], ad injection [126, 127, 128], survey scams [129, 119], cross-site scripting [130, 131, 132, 133], PHP code and SQL injection [134, 135, 136, 133, 137, 138], file inclusion attacks [139, 140], dictionary attacks [146], etc. Substantial research on malicious advertisements has focused on isolation and containment [3,15,34]. Other approaches have focused on detecting drive-by downloads by employing the properties of HTTP redirections to identify malicious behavior

[38,45]. Dynamic analyses have also been used to detect drive-by downloads and web-hosted malware [11,12,36]. Li et al. [147] investigated the advertisement delivery process to detect malvertising by automatically generating detection rules. Web tripwires were proposed to detect in-flight page changes performed by ISPs to inject advertisements.

Their adoption by website operators to detect malicious CMS plugins is limited by significant instrumentation and training complexities associated with these techniques. Conversely, this research proposes an automated investigation framework, agnostic to targeted attack classes, and can be deployed by all stakeholders in the CMS ecosystem.

5.5 Measurement Studies

WordPress plugin research focused on measuring vulnerabilities [148, 149, 150] and comparing plugin ratings with vulnerability exploits [151]. Researchers also assessed the role of web hosting providers to detect compromised websites [152], studied malicious web apps [35], malicious browser extensions [36, 37], and malicious packages in package registries [38]. Caballero et. al. [116] measured pay-per-install malware distribution in benign software. Meiser et al. [153] studied the cross-origin data exchange practices of 5k websites to assess the extent to which their security could be affected by the presence of an XSS vulnerability on one of their communication partners.

Chen et al. [154] performed a large-scale measurement of CORS misconfigurations. Among the 480k domains that they analyzed, they discovered that 27.5% of them are affected by some vulnerability and, in particular, 84k trust all their subdomains and can thus be exploited by a related-domain attacker. Son and Shmatikov [155] analyzed the usage of the Messaging API on the top 10k Alexa websites. The authors found that 1.5k hosts do not perform any origin checking on

the receiving message, while 261 implement an incorrect check: (almost) all these checks can be bypassed from a related-domain position, although half of them can also be bypassed from domains with a specially-crafted name. More recently, Steffens and Stock [156] proposed an automated framework for the analysis of postMessage handlers and used it to perform a comprehensive analysis of the first top 100k websites of the Tranco list. They discovered 111 vulnerable handlers, out of which 80 do not perform any origin check. Regarding the remaining handlers, the authors identified only 8 incorrect origin validations, showing an opposite trend with respect to [155]. Finally, insecure configurations of CSP have been analyzed in a number of research papers [157, 158]. However, unlike this research, none of these works considered the impact of scalable attacks originating from unvetted code on CMS marketplaces.

CHAPTER 6
CONCLUSION

In this dissertation, I have presented a line of research which has proposed a paradigm shift in server-side security practices. My work has purposely broken away from traditional log-based provenance inference, and instead I have developed a web attack forensics framework which leverages program analysis to automatically understand the webserver's nightly backup snapshots. In doing so, this framework has enabled the recovery temporal phases of a webserver compromise and its origin within the website supply chain. These three techniques along with the new program analysis techniques which enable them, have introduced new packing-oblivious forensics capabilities far exceeding traditional provenance inference.

Targeting the problem of investigating compromises in CMS-based websites using *only* the readily available nightly backups, TARDIS provides a novel provenance inference technique that reconstructs the attack phases and enables rapid recovery from an attack. Using the temporal correlation of spatial metrics representing each snapshot, TARDIS recovers the compromise window and the progression of attack phases.

YODA contributed to the novel web attack forensics techniques at a supply chain level and provided an automated investigation framework that can be used by all stakeholders in the webserver ecosystem. It identifies attack behaviors in malicious plugin and pinpoints the origin of these behaviors within the webserver supply chain.

Finally, to address the real-world webserver forensics challenge of code obfuscation and packing, I presented OBIWAN. Instead of focusing on statically recovering the attack behaviors, OBIWAN uses guided unpacking to generate the layers of hidden code. It also categorizes the obfuscation and packing types which helps understand

the attack techniques used over the last decade.

My experiments show that TARDIS is able to effectively identify the compromise window in a variety of real-world production websites overcoming the long standing provenance inference challenge. In fact, it uncovered 20,591 websites that were victims of long-lived multi-stage attacks and was shown to be highly accurate in revealing attacks in CMS-based websites, regardless of the underlying CMS. My tests with YODA produced a systematic study of the CMS plugin ecosystem by analyzing 410,122 unique WordPress websites' nightly backups dating back to 2012. It uncovered 47,337 malicious plugin installs on 24,931 unique websites, and 94% of these malicious plugins installed over those the last decade *are still active today*. OBIWAN uncovered over 10.1M obfuscated malware across over 27K websites, and 8.7M of these malware used packing. My research highlighted that the existing defenses are insufficient and are enabling attackers to retaliate with simple defenses. In particular, only 11% of all packed files are detected by AVs, and over 60% of the unpacked malware payloads in our dataset were reused.

In conclusion, the robust, packing-oblivious forensics capabilities realized by this new web attack forensics analysis framework highlight the impactful benefit and possibilities of program-analysis-driven forensics techniques.

REFERENCES

[1] *Is WordPress Really A 10 Billion Dollar Economy?* https://www.presstitan.com/is-wordpress-really-a-10-billion-dollar-economy/, [Accessed: 2020-05-08].

[2] S. M. Milajerdi, R. Gjomemo, B. Eshete, R. Sekar, and V. Venkatakrishnan, "HOLMES: Real-time apt detection through correlation of suspicious information flows," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2019.

[3] S. Ma, J. Zhai, F. Wang, K. H. Lee, X. Zhang, and D. Xu, "MPI: Multiple perspective attack investigation with semantic aware execution partitioning," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[4] Y. Kwon, F. Wang, W. Wang, K. H. Lee, W.-C. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, A. Gehani, and V. Yegneswaran, "MCI: Modeling-based causality inference in audit logging for attack investigation," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[5] Y. Kwon, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu, "LDX: Causality inference by lightweight dual execution," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[6] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[7] K. H. Lee, X. Zhang, and D. Xu, "LogGC: Garbage collecting audit log," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[8] S. Ma, K. H. Lee, C. H. Kim, J. Rhee, X. Zhang, and D. Xu, "Accurate, low cost and instrumentation-free security audit logging for windows," in *Proceedings of the 31st Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, Dec. 2015.

[9] P. Vadrevu, J. Liu, B. Li, B. Rahbarinia, K. H. Lee, and R. Perdisci, "Enabling reconstruction of attacks on users via e□cient browsing snapshots," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[10] S. Ma, J. Zhai, Y. Kwon, K. H. Lee, X. Zhang, G. Ciocarlie, A. Gehani, V. Yegneswaran, D. Xu, and S. Jha, "Kernel-supported cost-e□ective audit logging for causality tracking," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[11] S. Ma, X. Zhang, and D. Xu, "ProTracer: Towards practical provenance tracing by alternating between logging and tainting," in *Proceedings of the 2016 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[12] S. Sitaraman and S. Venkatesan, "Forensic analysis of file system intrusions using improved backtracking," in *Proceedings of the 3rd IEEE International Workshop on Information Assurance*, IEEE, College Park, MD, USA, Mar. 2005.

[13] K. Pei, Z. Gu, B. Saltaformaggio, S. Ma, F. Wang, Z. Zhang, L. Si, X. Zhang, and D. Xu, "HERCULE: attack story reconstruction via community discovery on correlated log graph," in *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, Los Angeles, CA, Dec. 2016.

[14] F. Wang, Y. Kwon, S. Ma, X. Zhang, and D. Xu, "Lprov: Practical library-aware provenance tracing," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

[15] M. N. Hossain, S. M. Milajerdi, J. Wang, B. Eshete, R. Gjomemo, R. Sekar, S. D. Stoller, and V. Venkatakrishnan, "Sleuth: Real-time attack scenario reconstruction from cots audit data," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Canada, Aug. 2017.

[16] Y. Liu, M. Zhang, D. Li, K. Jee, Z. Li, Z. Wu, J. Rhee, and P. Mittal, "Towards a timely causality analysis for enterprise security," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[17] *W3Techs - Usage of content management systems for websites*, https://w3techs.com/technologies/overview/content_management/all, [Accessed: 2019-01-16].

[18] *Popular CMS by Market Share*, https://websitesetup.org/popular-cms/, [Accessed: 2019-06-30].

[19] *HostGator.com LLC*, https://www.hostgator.com, [Accessed: 2019-06-12].

[20] *Dropmysite - Cloud Backups for Websites & Databases*, https://www.dropmysite.com/, [Accessed: 2018-10-31].

[21] *CodeGuard*, https://www.codeguard.com/, [Accessed: 2019-01-20].

[22] *GoDaddy*, https://www.godaddy.com/web-security/website-backup, [Accessed: 2018-01-20].

[23] *Sucuri*, https://sucuri.net/website-backups/, [Accessed: 2018-10-31].

[24] *iPage*, https://www.ipage.com/web-backup, [Accessed: 2018-10-31].

[25] W. U. Hassan, S. Guo, D. Li, Z. Chen, K. Jee, Z. Li, and A. Bates, "Nodoze: Combatting threat alert fatigue with automated provenance triage," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[26] *New Research from Advanced Threat Analytics*, https://www.prnewswire.com/news-releases/new-research-from-advanced-threat-analytics-finds-mssp-incident-responders-overwhelmed-by-false-positive-security-alerts-300596828.html, [Accessed: 2019-01-20].

[27] D. Canali and D. Balzarotti, "Behind the scenes of online attacks: An analysis of exploitation behaviors on the web," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[28] *zxcvbn: Low-Budget Password Strength Estimation*, https://github.com/dropbox/zxcvbn, [Accessed: 2019-05-28].

[29] D. C. Howell, "Median absolute deviation," *Wiley StatsRef: statistics reference online*, 2014.

[30] A. Koufakou, E. G. Ortiz, M. Georgiopoulos, G. C. Anagnostopoulos, and K. M. Reynolds, "A scalable and e□cient outlier detection strategy for categorical data," in *19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2007)*, IEEE, vol. 2, 2007, pp. 210–217.

[31] *What CMS Is This Site Using?* https://whatcms.org/, [Accessed: 2019-06-26].

[32] *CMS-Garden CMSScanner*, https://github.com/CMS-Garden/cmsscanner, [Accessed: 2019-06-12].

[33] *Pandas: Flexible and powerful data analysis and manipulation library for Python*, https://github.com/pandas-dev/pandas, [Accessed: 2019-05-28].

[34] *Drupal: CVE-2018-7600: Remote Code Execution - SA-CORE-2018-002*, https://www.rapid7.com/db/vulnerabilities/drupal-cve-2018-7600, [Accessed: 2019-06-26].

[35] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang, "Hey, you, get o□ of my market: Detecting malicious apps in o□cial and alternative android markets.," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[36] N. Jagpal, E. Dingle, J.-P. Gravel, P. Mavrommatis, N. Provos, M. A. Rajab, and K. Thomas, "Trends and lessons from three years fighting malicious extensions," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[37] A. Kapravelos, C. Grier, N. Chachra, C. Kruegel, G. Vigna, and V. Paxson, "Hulk: Eliciting malicious behavior in browser extensions," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[38] R. Duan, O. Alrawi, R. Pai Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards Measuring Supply Chain Attacks on Package Managers," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Conference, Feb. 2021.

[39] K. Soska and N. Christin, "Automatically detecting vulnerable websites before they turn malicious," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[40] J. Dahse and T. Holz, "Static detection of second-order vulnerabilities in web applications," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[41] L. Invernizzi, P. M. Comparetti, S. Benvenuti, C. Kruegel, M. Cova, and G. Vigna, "Evilseed: A guided approach to finding malicious web pages," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2012.

[42] R. P. Kasturi, Y. Sun, R. Duan, O. Alrawi, E. Asdar, V. Zhu, Y. Kwon, and B. Saltaformaggio, "TARDIS: Rolling Back The Clock On CMS-Targeting Cyber Attacks," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P)*, Virtual Conference, May 2020.

[43] *Is That WordPress Plugin Safe? 15 Warning Signs to Skip Downloading*, https://premium.wpmudev.org/blog/is-that-wordpress-plugin-safe-15-warning-signs-to-skip-downloading/, [Accessed: 2020-08-19].

[44] *WPScan Vulnerability Database*, https://wpvulndb.com/plugins, [Accessed: 2020-05-09].

[45] *RIPSTECH Vulnerability Detection*, https://www.ripstech.com/, [Accessed: 2020-05-09].

[46] *WebARX: Protect websites from plugin vulnerabilities*, https://www.webarxsecurity.com/, [Accessed: 2020-05-09].

[47] *9 WordPress Plugins Targeted in Coordinated 4.5-Year Spam Campaign*, https://www.wordfence.com/blog/2017/09/coordinated-plugin-spam/, [Accessed: 2020-05-08].

[48] *WP-VCD: The Malware You Installed On Your Own Site*, https://www.wordfence.com/wp-content/uploads/2019/11/Wordfence-WP-VCD-Whitepaper.pdf, [Accessed: 2020-06-27].

[49] *WordPress Themes*, https://wordpress.org/plugins/, [Accessed: 2020-08-08].

[50] *WordPress Themes*, https://wordpress.org/themes/, [Accessed: 2020-08-08].

[51] *Github*, https://github.com/, [Accessed: 2020-08-08].

[52] *CodeCanyon*, https://codecanyon.net/, [Accessed: 2020-08-08].

[53] *ThemeForest*, https://themeforest.net/, [Accessed: 2020-08-08].

[54] *WPMU DEV*, https://premium.wpmudev.org/, [Accessed: 2020-08-08].

[55] *Easy Digital Downloads*, https://easydigitaldownloads.com/, [Accessed: 2020-08-08].

[56] *The 10 Million Dollar Plugin: Choco Drops + Admob (Android Studio + Eclipse) Easy Reskin*, https://codecanyon.net/item/choco-drops-admob-android-studio-eclipse-easy-reskin/21423129, [Accessed: 2020-07-29].

[57] *PLR Products*, https://www.plrproducts.com/keyword-swarm-wp-plugin/, [Accessed: 2020-08-18].

[58] *Jeroen Sormani - WordPress Plugin Developer*, https://jeroensormani.com/, [Accessed: 2020-08-20].

[59] *VaultPress - Daily and Real-time Backups*, https://vaultpress.com/, [Accessed: 2020-08-20].

[60] *WordPress File Permissions: Complete Beginner's Guide*, http://www.themeshunter.com/stats/themeforest-net.html, [Accessed: 2021-09-09].

[61] *Why You Should Stop Using Nulled WordPress Plugins and Themes*, https://kinsta.com/blog/nulled-wordpress-plugins-themes/, [Accessed: 2020-10-07].

[62] *WordPress File Permissions: Complete Beginner's Guide*, https://www.malcare.com/blog/wordpress-file-permissions/, [Accessed: 2021-06-03].

[63] M. Backes, S. Bugiel, and E. Derr, "Reliable third-party library detection in android and its security applications," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[64] *Display Widgets Plugin Permanently Removed from WordPress.org Due to Malicious Code*, https://wptavern.com/display-widgets-plugin-permanently-removed-from-wordpress-org-due-to-malicious-code, [Accessed: 2020-10-12].

[65] *Abandoned and Removed Plugin Alerts*, https://www.wordfence.com/blog/2017/06/abandoned-removed-plugin-alerts/, [Accessed: 2020-10-12].

[66] *SweetCAPTCHA Service Used to Distribute Adware*, https://blog.sucuri.net/2015/06/sweetcaptcha-service-used-to-distribute-adware.html, [Accessed: 2020-10-12].

[67] Y. Li, J. Huang, A. Ikusan, M. Mitchell, J. Zhang, and R. Dai, "Shellbreaker: Automatically detecting php-based malicious web shells," *Computers & Security*, vol. 87, p. 101595, 2019.

[68] H. Cui, D. Huang, Y. Fang, L. Liu, and C. Huang, "Webshell detection based on random forest–gradient boosting decision tree algorithm," in *2018 IEEE Third International Conference on Data Science in Cyberspace (DSC)*, IEEE, 2018, pp. 153–160.

[69] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, M. Bagheri-Marzijarani, and J. W. Davidson, "Cubismo: Decloaking server-side malware via cubist program analysis," in *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, San Juan, Puerto Rico, Dec. 2019.

[70] N. Popov, *A PHP parser written in PHP*, Jul. 2019.

[71] V. Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www.virustotal.com/en*, 2012.

[72] *URLHaus online virus, malware and url scanner, author=URLHaus*, https://www.urlhaus.com, [Accessed: 2021-26-04].

[73] *Flatsome | Multi-Purpose Responsive WooCommerce Theme*, https://themeforest.net/item/flatsome-multipurpose-responsive-woocommerce-theme/5484319, [Accessed: 2020-09-23].

[74] *WPBakery Page Builder for WordPress*, https://codecanyon.net/item/visual-composer-page-builder-for-wordpress/242431, [Accessed: 2020-09-23].

[75] *FormCraft - Premium WordPress Form Builder*, https://codecanyon.net/item/formcraft-premium-wordpress-form-builder/5335056, [Accessed: 2020-09-23].

[76] *Gravity Forms*, https://www.gravityforms.com/pricing/, [Accessed: 2020-09-23].

[77] *WP Robot 5 - Your Blog On Autopilot*, https://wprobot.net/order/, [Accessed: 2020-09-23].

[78] *BeTheme - Responsive Multi-Purpose WordPress Theme*, https://themeforest.net/item/betheme-responsive-multipurpose-wordpress-theme/7758048, [Accessed: 2020-09-23].

[79] *WoodMart - Responsive WooCommerce WordPress Theme*, https://themeforest.net/item/woodmart-woocommerce-wordpress-theme/20264492, [Accessed: 2020-09-23].

[80] *DooPlay Theme WordPress*, https://www.downloadfreethemes.co/dooplay-v2-1-3-8-movies-and-tv-shows-wordpress-theme/, [Accessed: 2020-06-27].

[81] *Internet Archive: Wayback Machine*, https://archive.org/web/, [Accessed: 2021-05-10].

[82] P. Wrench and B. Irwin, "A sandbox-based approach to the deobfuscation and dissection of php-based malware," *Saiee Africa Research Journal*, 2015.

[83] *W3 Techs: Usage statistics of content management systems*, https://w3techs.com/technologies/overview/content_management, [Accessed: 2022-05-04].

[84] R. P. Kasturi, J. Fuller, Y. Sun, O. Chabklo, A. Rodriguez, J. Park, and B. Saltaformaggio, "Mistrust plugins you must: A large-scale study of malicious plugins in wordpress marketplaces," in *Proceedings of the 31st USENIX Security Symposium (Security)*, Boston, MA, Aug. 2022.

[85] C. C. B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Low-overhead mostly static javascript malware detection," in *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[86] D. Canali, M. Cova, G. Vigna, and C. Kruegel, "Prophiler: A fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th International World Wide Web Conference (WWW)*, Hyderabad, India, Apr. 2011.

[87] C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert, "Rozzle: De-cloaking internet malware," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2012.

[88] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[89] A. Kapravelos, Y. Shoshitaishvili, M. Cova, C. Kruegel, and G. Vigna, "Revolver: An automated approach to the detection of evasive web-based malware," in *Proceedings of the 22th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[90] L. Lu, V. Yegneswaran, P. Porras, and W. Lee, "Blade: An attack-agnostic approach for preventing drive-by malware infections," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Oct. 2010.

[91] M. Van Gundy, D. Balzarotti, and G. Vigna, "Catch me, if you can: Evading network signatures with web-based polymorphic worms.."

[92] O. Starov, J. Dahse, S. S. Ahmad, T. Holz, and N. Nikiforakis, "No honor among thieves: A large-scale analysis of malicious web shells," in *Proceedings of the 25th International World Wide Web Conference (WWW)*, 2016.

[93] *UnPHP - The Online PHP Decoder*, https://www.unphp.net/, [Accessed: 2022-05-02].

[94] *Top PHP Security and Malware Scanners*, https://phpmagazine.net/2020/10/top-php-security-and-malware-scanners.html, [Accessed: 2022-05-05].

[95] A. Naderi-Afooshteh, Y. Kwon, A. Nguyen-Tuong, A. Razmjoo-Qalaei, M.-R. Zamiri-Gourabi, and J. W. Davidson, "Malmax: Multi-aspect execution for automated dynamic web server malware analysis," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2011.

[96] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: Fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2006.

[97] *The Hacker Motive: What Attackers Are Doing with Your Hacked Site*, https://www.wordfence.com/blog/2020/09/the-hacker-motive-what-attackers-are-doing-with-your-hacked-site/, [Accessed: 2022-05-05].

[98] *VirusTotal: Analyze suspicious files, domains, IPs and URLs to detect malware and other breaches, automatically share them with the security community*, https://www.virustotal.com/, [Accessed: 2022-05-03].

[99] A. Mantovani, S. Aonzo, X. Ugarte-Pedrero, A. Merlo, and D. Balzarotti, "Prevalence and impact of low-entropy packing schemes in the malware ecosystem.," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[100] A. Fass, M. Backes, and B. Stock, "Hidenoseek: Camouflaging malicious javascript in benign asts," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2011.

[101] X. Ugarte-Pedrero, D. Balzarotti, I. Santos, and P. G. Bringas, "Sok: Deep packer inspection: A longitudinal study of the complexity of run-time packers," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2015.

[102] *Zend Guard Obfuscator and Encoder*, https://www.zend.com/products/zend-guard, [Accessed: 2022-04-10].

[103] *Zend Guard Obfuscator and Encoder*, https://www.ioncube.com/php_encoder.php, [Accessed: 2022-04-10].

[104] *CMS-Garden CMSScanner*, https://github.com/CMS-Garden/cmsscannerl, [Accessed: 2022-04-09].

[105] O. Catakoglu, M. Balduzzi, and D. Balzarotti, "Automatic extraction of indicators of compromise for web applications," in *Proceedings of the 25th International World Wide Web Conference (WWW)*, 2016.

[106] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *Proceedings of the 22nd International World Wide Web Conference (WWW)*, Rio de Janeiro, Brazil, May 2013.

[107] *Google Hack Honeypot*, http://ghh.sourceforge.net/, [Accessed: 2021-06-03].

[108] *MushMush Foundation*, http://mushmush.org/, [Accessed: 2021-06-03].

[109] J. P. John, F. Yu, Y. Xie, A. Krishnamurthy, and M. Abadi, "Heat-seeking honeypots: Design and experience," in *Proceedings of the 20th International World Wide Web Conference (WWW)*, Hyderabad, India, Apr. 2011.

[110] B. Stock, G. Pellegrino, C. Rossow, M. Johns, and M. Backes, "Hey, you have a problem: On the feasibility of large-scale web vulnerability notification," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[111] "To Get Lost is to Learn the Way: Automatically Collecting Multi-step Social Engineering Attacks on the Web," in *Proceedings of the 15th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Taipei, Taiwan, Oct. 2020.

[112] T. Nelms, R. Perdisci, M. Antonakakis, and M. Ahamad, "Webwitness: Investigating, categorizing, and mitigating malware download paths," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[113] N. Provos, D. McNamee, P. Mavrommatis, K. Wang, N. Modadugu, *et al.*, "The ghost in the browser: Analysis of web-based malware.," *HotBots*, vol. 7, pp. 4–4, 2007.

[114] A. Sudhodanan, S. Khodayari, and J. Caballero, "Cross-Origin State Inference (COSI) Attacks: Leaking Web Site States through XS-Leaks," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[115] C.-A. Staicu and M. Pradel, "Leaky images: Targeted privacy attacks in the web," in *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[116] J. Caballero, C. Grier, C. Kreibich, and V. Paxson, "Measuring pay-per-install: The commoditization of malware distribution.," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[117] B. J. Kwon, J. Mondal, J. Jang, L. Bilge, and T. Dumitraş, "The dropper e□ect: Insights into malware distribution with downloader graph analytics," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[118] P. Vadrevu, B. Rahbarinia, R. Perdisci, K. Li, and M. Antonakakis, "Measuring and detecting malware downloads in live network tra□c."

[119] J. W. Clark and D. McCoy, "There are no free ipads: An analysis of survey scams as a business.," in *Proceedings of the 6th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Washington, D.C., United States, Aug. 2013.

[120] F. Li, Z. Durumeric, J. Czyz, M. Karami, M. Bailey, D. McCoy, S. Savage, and V. Paxson, "You've got vulnerability: Exploring e□ective vulnerability notifications," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[121] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[122] K.-K. Muniswamy-Reddy, U. Braun, D. A. Holland, P. Macko, D. L. MacLean, D. W. Margo, M. I. Seltzer, and R. Smogor, "Layering in provenance systems.," in *Proceedings of the 2009 USENIX Annual Technical Conference (ATC)*, San Diego, CA, Jun. 2009.

[123] L. Akoglu, H. Tong, and D. Koutra, "Graph based anomaly detection and description: A survey," *Data mining and knowledge discovery*, 2015.

[124] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer, "Trustworthy whole-system provenance for the linux kernel," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[125] T. Pasquier, X. Han, T. Moyer, A. Bates, O. Hermant, D. Eyers, J. Bacon, and M. Seltzer, "Runtime analysis of whole-system provenance," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Canada, Oct. 2018.

[126] S. Arshad, A. Kharraz, and W. Robertson, "Identifying extension-based ad injection via fine-grained web content provenance," in *Proceedings of the 19th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Evry, France, Sep. 2016.

[127] K. Thomas, E. Bursztein, C. Grier, G. Ho, N. Jagpal, A. Kapravelos, D. McCoy, A. Nappa, V. Paxson, P. Pearce, N. Provos, and M. Abu Rajab, "Ad injection at scale: Assessing deceptive advertisement modifications," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2015.

[128] X. Xing, W. Meng, B. Lee, U. Weinsberg, A. Sheth, R. Perdisci, and W. Lee, "Understanding malvertising through ad-injecting browser extensions," in *Proceedings of the 24th International World Wide Web Conference (WWW)*, Florence, Italy, 2015.

[129] A. Kharraz, W. Robertson, and E. Kirda, "Surveylance: Automatically detecting online survey scams," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, May 2018.

[130] W. Melicher, A. Das, M. Sharif, L. Bauer, and L. Jia, "Riding out domsday: Toward detecting and preventing dom cross-site scripting," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[131] B. Stock, S. Lekies, T. Mueller, P. Spiegel, and M. Johns, “Precise client-side protection against dom-based cross-site scripting.,” in *Proceedings of the 2014 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[132] G. Wassermann and Z. Su, “Static detection of cross-site scripting vulnerabilities,” in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig, Germany, May 2008.

[133] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, “E□cient and flexible discovery of php application vulnerabilities,” in *Proceedings of the European Symposium on Security and Privacy (EuroS&P)*, Paris, France, Apr. 2017.

[134] D. R. Sahu and D. S. Tomar, “DNS pharming through PHP injection: Attack scenario and investigation,” *IJ Computer Network and Information Security*, vol. 4, pp. 21–28, 2015.

[135] V. Yerram and G. V. R. Reddy, “A solution to php code injection attacks and web vulnerabilities,” 2014.

[136] Z. S. Alwan and M. F. Younis, “Detection and prevention of sql injection attack: A survey,” *International Journal of Computer Science and Mobile Computing*, vol. 6, no. 8, pp. 5–17, 2017.

[137] N. Singh, M. Dayal, R. Raw, and S. Kumar, “Sql injection: Types, methodology, attack queries and prevention,” in *Proceedings of the 3rdComputing for Sustainable Global Development (INDIACom)*, IEEE, New Delhi, India, Mar. 2016.

[138] A. Pramod, A. Ghosh, A. Mohan, M. Shrivastava, and R. Shettar, “Sqli detection system for a safer web application,” in *Proceedings of the 2015IEEE International Advance Computing Conference*, IEEE, Bangalore, India, Jun. 2015.

[139] H. F. G. Robledo, “Types of hosts on a remote file inclusion (rfi) botnet,” in *Proceedings of the Electronics, Robotics and Automotive Mechanics Conference*, Jun. 2008.

[140] O. Katz, “Detecting remote file inclusion attacks,” *White Paper. Breach Security Inc., May*, 2009.

[141] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities,” in *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2006.

[142] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Cross site scripting prevention with dynamic data tainting and static analysis..”

[143] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, “Saner: Composing static and dynamic analysis to validate sanitization in web applications,” in *Proceedings of the 29th IEEE Symposium on Security and Privacy (S&P)*, Oakland, CA, May 2008.

[144] T. Scholte, W. Robertson, D. Balzarotti, and E. Kirda, “Preventing input validation vulnerabilities in web applications through automated type analysis,” in *Computer software and applications conference*, 2012.

[145] S. McAllister, E. Kirda, and C. Kruegel, "Leveraging user interactions for in-depth testing of web applications," in *Proceedings of the 11th International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, Cambridge, Massachusetts, Sep. 2008.

[146] A. K. Kyaw, F. Sioquim, and J. Joseph, "Dictionary attack on wordpress: Security and forensic analysis," in *2015 Second International Conference on Information Security and Cyber Forensics (InfoSec)*, 2015.

[147] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang, "Knowing your enemy: Understanding and detecting malicious web advertising," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[148] J. Ruohonen, "A demand-side viewpoint to software vulnerabilities in wordpress plugins," in *Evaluation and Assessment on Software Engineering*, 2019.

[149] O. Mesa, R. Vieira, M. Viana, V. H. Durelli, E. Cirilo, M. Kalinowski, and C. Lucena, "Understanding vulnerabilities in plugin-based web systems: An exploratory study of wordpress," in *22nd International Systems and Software Product Line Conference*, 2018.

[150] I. Cernica, N. Popescu, and B. ţigănoaia, "Security evaluation of wordpress backup plugins," in *2019 22nd International Conference on Control Systems and Computer Science (CSCS)*, 2019.

[151] T. Koskinen, P. Ihantola, and V. Karavirta, "Quality of wordpress plug-ins: An overview of security and user ratings," in *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Confernece on Social Computing*, 2012.

[152] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *Proceedings of the 22nd International World Wide Web Conference (WWW)*, Rio de Janeiro, Brazil, May 2013.

[153] G. Meiser, P. Laperdrix, and B. Stock, "Careful who you trust: Studying the pitfalls of cross-origin communication," in *Proceedings of the 16th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Hong Kong, China, Jun. 2021.

[154] J. Chen, J. Jiang, H. Duan, T. Wan, S. Chen, V. Paxson, and M. Yang, "We still don't have secure cross-domain requests: An empirical study of cors," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[155] S. Son and V. Shmatikov, "The postman always rings twice: Attacking and defending postmessage in html5 websites.," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[156] M. Ste□ens and B. Stock, "Pmforce: Systematically analyzing postmessage handlers at scale," in *Proceedings of the 27th ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Nov. 2020.

[157] S. Roth, T. Barron, S. Calzavara, N. Nikiforakis, and B. Stock, "Complex security policy? a longitudinal analysis of deployed content security policies," in *Proceedings of the 2020 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2020.

[158] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "Csp is dead, long live csp! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.