Stealthy server-side attacks using benign websites and a Fail-free Dynamic State Machine

---

A

Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment

of the requirements for the degree

Master of Science

by

Bora Lee

April 2023

APPROVAL SHEET

This

Thesis

is submitted in partial fulfillment of the requirements
for the degree of

Master of Science

Author: Bora Lee

This Thesis has been read and approved by the examing committee:

Advisor: Dr. Yonghwi Kwon

Advisor:

Committee Member: Dr. Yixin Sun

Committee Member: Dr. Tianhao Wang

Committee Member:

Committee Member:

Committee Member:

Committee Member:

Accepted for the School of Engineering and Applied Science:

Jennifer L. West, School of Engineering and Applied Science

April 2023

Abstract

The presence of server-side malware poses a significant risk to a large number of clients who access the compromised server. In this research, we propose a Stealthy-Attack on the server-side that can withstand forensic analysis such as reverse-engineering. Our attack can be triggered by ordinary contents from legitimate and benign websites to avoid detection and misdirect investigators. To expand the input-output space and make reverse-engineering challenging, our attack uses a specialized state-machine that accepts any inputs and produces output accordingly. We created a prototype of Stealthy-Attack and conducted an empirical evaluation on the attack, which demonstrates that it poses significant obstacles to forensic analysis.

Contents

Chapter 1

Introduction

1.1 Background

Malware analysis is a crucial task in revealing the real intentions and actors behind cyber attacks. Analyzing malware can lead to various forensic evidence, such as what sensitive information the malware wants to leak and to where (e.g., addresses of attacker-controlled servers). In recent years, malware gets more sophisticated in hiding its code using various techniques such as code obfuscation and remote code execution. In response, advanced malware analysis techniques have been proposed: (1) program analysis techniques including symbolic execution and forced execution [13, 17, 19, 23, 27, 28, 31, 35, 37, 40, 48, 51, 61, 62, 63, 64, 65, 71, 84, 89] that can uncover hidden malicious logic in malware and (2) deep packet inspection techniques [9, 20, 42, 74] that can see through malicious payloads delivered through network packets. While it is challenging to dissect malware completely, analyzing behaviors of malware often results in critical hints for triaging the attacker (e.g., via network addresses they connect to).

In this research, we explore the possibility of creating a forensically stealthy malware. Specifically, we present an anti-forensic attack, dubbed Stealthy-Attack. It collects inputs from multiple *benign and uncompromised websites* that are *not associated with cyber attackers* (e.g., www.npr.org). The input content is ordinary (i.e., not

influenced by the attacker), avoiding detection from network packet inspection techniques and leaving no forensic evidence in the network trace. Stealthy-Attack does not include executable malicious code itself, making static analysis based forensic analysis (e.g., anti-virus techniques) ineffective.

The inputs are later used to construct a malicious payload through a special state machine proposed by [33, 34], which is carefully designed to make the analysis of Stealthy-Attack inconclusive. Specifically, the state-machine can take any inputs and generate varying outputs depending on inputs, making the input-output space extremely large. As such, Stealthy-Attack evades state-of-the-art malware detection techniques, including reverse-engineering and forensic triaging. We design and implement a set of tools that can create Stealthy-Attack from the following two inputs: (1) malicious code snippet to deliver and (2) a set of benign contents. The created Stealthy-Attack will run the specialized state-machine to convert the predetermined benign contents into the malicious code snippet when all the benign contents appear together.

Our contributions are summarized as follows:

- We propose Stealthy-Attack, an anti-forensic technique that transforms ordinary contents from benign websites to malicious payloads.
- We leverage the concept of ambiguous translator [33, 34] for translating input words to malicious payloads to impose challenges in reverse-engineering.
- We implement a set of automated tools to create Stealthy-Attack, including a website crawler, statistical analyzer, and the ambiguous translator generator.
- Our evaluation result shows that Stealthy-Attack is effective in delivering malicious payloads without being analyzed and detected.

Threat model. We assume a forensic/malware analysis scenario. Specifically, we assume that an attacker already compromised a victim server and placed the malware. While the malware might be executed, it did not deliver the malicious payload (i.e., attack) yet. Exploiting servers can be done by leveraging software vulnerabilities [21, 67] in Internet-facing server programs. The exploitation of web servers is out of the scope of this research, but is typical in advanced cyber attack scenarios [2, 24, 36, 45]. We assume that the victim server may log network requests, but may not know when the malware delivered the malicious payload.

Chapter 2

Related Work

2.1 Obfuscation techniques

Previous research has explored methods of obfuscation that use opaque predicates to conceal malicious code[16, 19, 54], code insertion/replacement[22, 46], encryptions [72, 86], hardware primitives [14, 70]. Although some obfuscation techniques, such as the use of opaque predicates, have been used to conceal malicious code, they are not foolproof as they can still be detected and eliminated using advanced program analysis techniques [52].

Dummy code snippets that are inserted into an existing program can be detected and eliminated through the use of dependency analysis techniques, such as taint analysis [17, 18, 19, 26, 27, 28, 31, 32, 40, 48, 51, 61, 62, 63, 64, 70, 84, 89]. Translating a program into a more abstracted form, such as AST or Intermediate Representation, can also reveal obfuscation techniques. A type of obfuscation known as data obfuscation, which involves encrypting code sections and decrypting them during runtime, can be easily detected using dynamic analysis techniques [12, 50, 74]. Approaches that rely on specific hardware support are difficult to use in real-world malware, as many systems may not meet the necessary hardware requirements. In contrast, the Stealthy-Attack does not require any specific hardware or software, making it particularly difficult to detect. Furthermore, the Stealthy-Attack is much more challenging

to analyze using static, symbolic, and dynamic analysis techniques, including fuzz testing tools, as demonstrated in Chapter 3

2.2 Advanced malware analysis techniques

A group of research [3, 4, 6, 15, 23, 32, 35, 38, 41, 46, 53, 68, 71, 75, 80] tries to detect and analyze malware. In particular, a dynamic analysis based forced execution technique [37] aims to handle evasive JavaScript malware. They forcibly drive execution into every branch even if the branch condition is not satisfied. While they are effective in detecting malware that hides malicious code behind sophisticated predicates, it is not effective in exposing malicious payload in Stealthy-Attack because it is encoded as states and transitions of the FDSM. Moreover, there are static, symbolic [26], and fuzzing tools for malware analysis that can reveal malicious behaviors. As discussed in Chapter 5, Stealthy-Attack is resilient to such malware analysis techniques.

2.3 Network traffic based analysis

There are also forensic analysis and network traffic analysis approaches that analyze the causal relationship between network and system events [4, 80]. For such techniques, Stealthy-Attack is difficult to analyze as it gets all inputs from common benign websites where many other applications and systems may access them when Stealthy-Attack is active. As a result, understanding who are the actors behind the attack is particularly challenging. There are approaches that detect common patterns of malware [35, 38]. While they are effective in traditional malware, Stealthy-Attack can evade such techniques as Stealthy-Attack can be implanted into existing programs.

Chapter 3

Motivating example

We show the effectiveness of the Stealthy-Attack by following a forensic analyst's perspective. Assume that a forensic analyst finds an instance of Stealthy-Attack in a compromised server[1], *before it launches the attack*. Then, he aims to understand (1) the purpose and (2) the actors behind the Stealthy-Attack. We present four different analysis attempts on Stealthy-Attack, to demonstrate its resilience to forensic analyses.

3.1 Analyzing inputs (i.e., network trace)

The forensic investigator obtains available network logs including network packet headers and actual payloads *before the attack happens*. Unfortunately, from the domain names, IP addresses, and the content that Stealthy-Attack has interacted with, the investigator cannot understand what it does. A naive way of attributing the attack to the benign websites, as shown in ❶ in Figure 3.1, is misleading the analysis.

3.2 Dynamic analysis

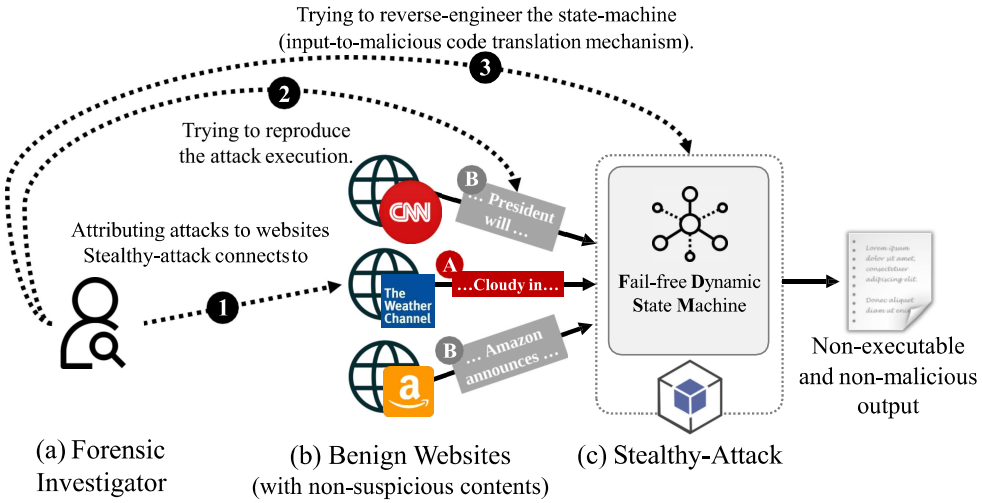
The analyst tries to execute the Stealthy-Attack sample, hoping that it can exercise intended malicious behaviors so that they can be analyzed (❷). However, without

[1]The assumption on the compromised servers in cyber attacks is typical [2, 24, 45].

knowing the particular input that can trigger the intended attack (which we call *attack triggering input*), the Stealthy-Attack instance does not expose its real intention (e.g., malicious code). Note that Stealthy-Attack will only generate malicious payloads when *the inputs from benign websites are presented as the attacker expected*. If not, it will generate a non-malicious output (i.e., a string that does not look any malicious or another malicious code for obscuring the real objective). For example, in Figure 3.1, Stealthy-Attack launches an attack (i.e., translates inputs to a malicious payload) when it receives "...Airlines say..." from *www.cnn.com*, "...Cloudy in..." from *www.weather.com*, and "...Amazon recommends..." from *www.amazon.com* as shown in the second row of Figure 3.1-(d). However, when the analyst executes the program, it obtains "...President will..." (B), "...Cloudy in..." (A), and "...Amazon announces..." (B), where A means that the input is a part of attack delivering input and B represents a non-attack delivering the input. As a result, the malicious payload (the third row of Figure 3.1-(d)) is not generated.

3.3 Static analysis

The analyst tries to use static analysis techniques including symbolic execution to learn the real intention of the Stealthy-Attack instance. However, they suffer from over-approximation. They may obtain a set of all possible inputs and outputs *without a particular order*, which cannot provide a concrete malicious code snippet. Moreover, among the identified outputs, there are no suspicious outputs (e.g., those look like code such as `unlink()`). This is because the state-machine can generate outputs that are different from the annotated outputs of the state machine. In other words, it can generate a malicious code snippet 'fwrite' without having the exact word 'fwrite' annotated in the state machine (Details are elaborated in Section 4.2.1). As a result,



Input	Output
"**Airlines say**", "**Cloudy in**", "**Amazon recommends**", ...	@fopen("tmp");@fwrite(...);...
"President will", "**Cloudy in**", "Amazon announces", ...	Invested @amazonnews more information ...

(d) Input Translation by Fail-free Dynamic State Machine

A Content that will deliver the attack B Content that will ***not*** deliver the attack

Figure 3.1: Forensic analysis on Stealthy-Attack. Malicious output is not generated without the attack-triggering input.

even after exploring millions of possible paths and inputs via symbolic execution tools [5, 23, 29, 57], the attack delivering inputs were not found (Details are described in Chapter 5).

3.4 Manual analysis

The analyst manually reads the source code to understand how the input words are translated and what the hidden malicious behaviors are (❸). The analyst collects all possible inputs that can be processed by the state machine and tries to construct inputs hoping it can reveal malicious payloads. However, he observes it is not possible to establish a one-to-one mapping because the same state transition can be triggered

by multiple inputs, which implies he may need to test almost every possible word (due to *dynamic output translation* in Section 4.2.1).

Chapter 4

Design

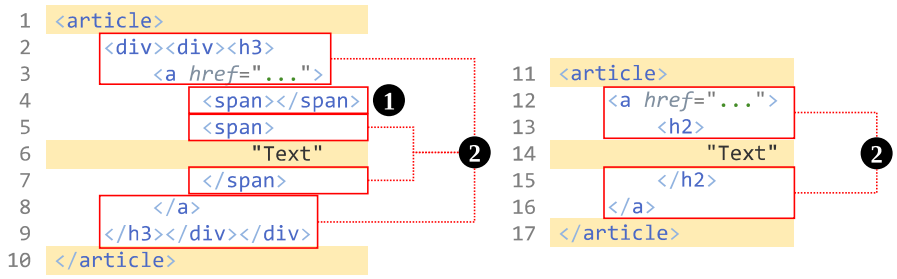
To create Stealthy-Attack, we first profile websites to identify candidate input contents (Section 4.1) and then construct the ambiguous translator (Section 4.2).

4.1 Identifying input words via profiling

Stealthy-Attack operates on the contents obtained from benign websites that are *not controllable* by attackers (e.g., a headline news title on *www.cnn.com*). This design choice is crucial to deceive forensic investigators (i.e., hide the identity of attackers). However, uncontrollable inputs might be unreliable because they may have changed when the attack is launched. We mitigate this issue by choosing inputs that are *statistically reliable* via website content profiling. In addition, we propose a DOM path update resilient parsing technique.

4.1.1 Profiling

The website profiler takes a set of web pages as input and generates multiple *InputVector* candidates, consisting of *Input Words* and *Statistics*. Input words are essentially the chosen inputs that make Stealthy-Attack launch the attack (i.e., deliver the malicious payload). It crawls the web pages regularly for a specified period (e.g., every



(a) Part of the persistent DOM tree of www.cnn.com (July 2019)

(b) Part of the persistent DOM tree of www.cnn.com (December 2020)

Figure 4.1: Comparing two persistent DOM trees.

hour for one month by default) so we have multiple snapshots for each page.

• **Obtaining a persistent DOM tree.** From the crawled website snapshots, we compute a *persistent* DOM tree that only includes elements that appear in *all crawled instances* so that unreliable contents will be excluded.

• **DOM parsing resilient to updates.** We leverage a parsing technique that is resilient to updates in websites' DOM structures. Specifically, we eliminate nodes that do not contain any text. In addition, for an element only contain another element (i.e., a holder element), we consider it can be reduced when it is compared. For instance, " <h3> Text </h3> " and " Text " are considered equivalent in the profiler and Stealthy-Attack. In addition, nodes with empty content will also be ignored as well (e.g., " "). These two techniques handle cosmetic changes in websites. Figure 4.1 shows an example of two persistent DOM trees obtained from www.cnn.com from (a) July 2019 and (b) December 2020. Observe that they have different DOM trees. However, after we remove tags that do not have any content (❶) and consider placeholders (❷) between the two DOM trees are equivalent, we identify the two are semantically identical (as the highlighted parts are same). This parsing technique is used by all the components of Stealthy-Attack.

• **Reliability of a persistent DOM tree.** To better understand the reliability of the persistent DOM tree, we obtain 5 persistent DOM trees collected from July 2019 (1st to 31st), November 2019 (1st to 30th), March 2020 (1st to 31st), July 2020 (1st to 31st), and November 2020 (1st to 30th) for 10 websites[1]. Then, we compare the five persistent DOM trees of each website. The result shows that 94.7% of the persistent DOM tree's elements reliably appear during the first 4 months. The percentage of reliable elements becomes smaller as time goes: 85.5% for 8 months, 76.3% for 12 months, and 71.3% for 16 months. This suggests that Stealthy-Attack is quite reliable in its first four months. Moreover, to improve the reliability further, we use multiple DOM elements and use them as alternative elements (i.e., backup options) as explained in Section 4.1.2.

• **Extracting input words.** For each DOM element in the persistent DOM tree, we extract all words that are longer than three characters. Shorter words (e.g., "for" and "or") are not good candidates in general because they usually do not have much meaning and are too frequently seen across different pages.

The extracted words are annotated with their positions found in the text of the DOM element. For example, a text "***Episode 976: Terms of Service***" is annotated as follows: "Episode" = 1st, "976:" = 2nd, "Terms" = 3rd, and "Service" = 4th. Observe that the word "of" is *not considered* as its length is *not longer than 3*. At runtime, the position will be used to extract input words.

• **Computing input word statistics.** Given the extracted words, the TextRank algorithm [56] is used to identify frequently observed words among them. The top ten (the number of words is configurable) words from the result are chosen. Then,

[1]www.cnn.com, www.npr.org, www.gnu.org, 19hz.info, techtonic.fm, earthquaketrack.com, news.ycombinator.com, www.kimbellart.org, lite.poandpo.com, chromereleases.googleblog.com

we compute the statistics of the chosen words. Specifically, for each input word, we calculate (1) coverage, (2) regularity, and (3) distribution of the word during the profiling period.

1) Coverage: This represents the percentage of an input word in a DOM element appearing during the profiling. For example, suppose that we crawl every hour for 5 days, resulting in 120 (=$24 * 5$) data points. If an input word appears 100 out of 120 data points, its coverage is $\frac{100}{120}$. Intuitively, an input word with a higher coverage has a higher probability of being observed in the future.

2) Regularity: The regularity represents how regularly the content will appear. To compute the regularity, for each input word, we measure the variance of time distances between the two adjacent appearances. Specifically, given N data points, the distances between the adjacent two points, n and n+1, are calculated, resulting N-1 distances: $d_1, d_2, ..., d_{N-1}$. Then, we count the number of unique distances, denoted as $CNT_{\text{unique\_distances}}$. If all the distances are equal (i.e., they regularly appear), the value (i.e., regularity) will be 1.0 (i.e., 100%). We compute the regularity as follows: $1.0 - \frac{CNT_{\text{unique\_distances}}}{N-1}$. Intuitively, an input word with a higher regularity will appear in a more predictable way than an input word with a lower regularity.

3) Distribution: It represents how an input word *appeared evenly* during the profiling period. This is complementary to the regularity because some input words with high regularity may not be evenly distributed. For instance, if a particular input word appears only three times but at the beginning, in the middle, and at the end of the profiling period, it would have a high regularity score. However, it is not well distributed over the period. A higher distribution value means an input word has been observed evenly and frequently during the profiling period. Algorithm 1 shows how we compute the distribution value (Dist). First, given a sequence of N data

Algorithm 1 Computing distribution

```
Input  : D: a set of data including all the profiled data points,
         N: the number of data points. W: input word.
         R: the number of iterations for distribution computation.
Output: Dist: distribution value for the input word W.
procedure Distribution(D, N, W)
    G ← 24, Dist ← 0, r ← 1
    while r ≠ R do
        i ← 0, D_c ← 0
        while i ≠ G do
            r ← N/G
            n ← i + 1
            D_sub ← D_[i*r, n*r]
            i ← i + 1
            if W ∈ D_sub then
                D_c ← D_c + 1
            end
        end
        Dist ← Dist + D_c / G
        G ← G * 2, r ← r + 1
    end
    return Dist / R
```

points (the input D), we divide them into G groups ($G = 24$ in this research, line 2) so that each group includes N/G data points (line 8 as D_{sub} represents the group). Then, we count whether the content appears in each group (line 10) and divide it by the value of G (lines 11-12). After that, we multiply G by 2 and repeat the above process (line 13). After we repeat this R times ($R = 5$ in this research), we add the computed value for different Gs and divide by R (line 14). To this end, an input word that appears in more groups will have a higher distribution value.

4.1.2 Handling unexpected DOM changes

A website may change its DOM structure. As Stealthy-Attack walks on a DOM tree to locate the input words, changes in the DOM structure can affect the input word collecting process.

To handle this problem, we *chain alternative input words* from multiple DOM elements so that Stealthy-Attack can reliably deliver the payload even with unexpected DOM changes. Specifically, for each selected input word, we identify input

words from *other DOM elements that always appear together* with the selected input word. The selected alternative DOM elements should *not share* many DOM elements in their DOM paths (e.g., no more than 20%) so that the alternative input words can work when a significant portion of DOM structure (e.g., 80% of the DOM structure) is changed. If there is no such alternative input word within the same webpage, we use input words from other pages. In practice, a website often has many duplicated contents across multiple sub-webpages. For instance, *www.npr.org*'s front page [59] and it's National News Section [60] have identical contents because top news in the "National News" section also appear in the front page. Such contents can be potential alternative input words to tolerate unexpected DOM changes. At runtime, if Stealthy-Attack fails to extract an input word from a webpage (e.g., from the front page) because the DOM element containing the input word cannot be located, it tries an alternative input word on another page (e.g., from the National News Section subpage). If it fails again, it continues trying the next alternative input words until it succeeds. In this research, we chain *4 alternative DOM elements* for an input word. The number of alternative DOM elements is configurable.

4.2 Stealthy-Attack Creator

Two inputs are required to create Stealthy-Attack: (1) chosen input words from the profiler and (2) payloads to deliver (e.g., source code of existing malware).

4.2.1 Fail-free Dynamic State Machine.

The core of Stealthy-Attack is the fail-free dynamic state machine (FDSM). The technique is borrowed from an existing work [34]. It has two distinctive anti-forensic

characteristics. First, it is a *fail-free* state machine that *always transits states* regardless of the current state and input, even if the input is *not annotated with the state transitions* (**C1**). In a typical state machine, a state transition only happens when there is a transition that can accept the current input. If not, the state machine will be stuck and fail to make a transition which can be traced by a forensic analyst to infer that the provided input is *not valid*. Second, the output generation rule of FDSM during state transition is *dynamic* (**C2**). This means that the output is changing based on a concrete input at runtime. This significantly enlarges the search space of the possible inputs and outputs.

• **Making transitions on any inputs (C1).** FDSM is designed to make transitions from any state on any inputs. If an input does not match with any possible transitions from the current state, it makes a transition to a state which has a transition condition most similar to the provided input. Specifically, for all next states from the current state, it calculates the distance (by subtracting values from each byte offset) between the current input and the transition conditions. Then, it selects a transition with the smallest distance.

• **Dynamic output translation (C2).** FDSM takes any inputs and generates outputs where each input leads to a *unique output*. When FDSM makes a transition on an input that is not exactly matched with the input of the transition, it changes the output translation rule by applying the differences between the current input and the input annotated on the transition. This makes the output space very large as the output can vary as much as the input varies.

Consider FDSM taking I as input and making a transition T^x where the transition's annotated input and output are denoted as T^x_{IN} and T^x_{OUT} respectively. Now, assume a scenario when an input with no matching transitions from the current state

is given. In this case, FDSM makes a transition T^x if the distance (i.e., the sum of the distance between characters) between I and T^x_{IN} is the smallest compare to other transitions' annotated inputs (i.e., T^{others}_{IN}). Moreover, we extend the output space by dynamically changing T^x_{OUT} based on the current input I. Specifically, given I that is different from T^x_{IN}, and assume that the state machine makes T^x transition, instead of generating T^x_{OUT} according to the state machine, we generate an output computed by $T^x_{OUT} - (T^x_{IN} - I)$ on each byte of T^x_{OUT}, T^x_{IN}, and I and '$-$' operator represents subtraction on each byte between the two operands (with the same byte offsets).

4.2.2 Constructing FDSM.

First, we create states and transitions for translating the chosen input to the given malicious payload, so that it can generate malicious payload when the predefined attack delivering inputs are provided. We then add dummy states and transitions to connect all states. Note that the dummy states and transitions can also be used to create decoy (i.e., fake) payloads so that it can mislead the forensic analysis. Inputs/outputs of the transitions to the dummy states are chosen in a way that the inputs of all transitions *look similar*, making it challenging to know which transitions are for malicious payload generations. Specifically, for each newly added transition, its input is derived by choosing a similar word (i.e., synonyms/antonyms in dictionaries [1, 66]) to its neighboring transition's input.

Chapter 5

Evaluation

5.1 Reliability of Stealthy-Attack

Stealthy-Attack takes input from webpages that are not under the control of the attacker, meaning that the reliability of Stealthy-Attack's attack is probabilistic. To understand the reliability of Stealthy-Attack in practice, we create a mock attack with real-world websites and show the result.

Experiment with real-world websites. To understand how reliably input words will show during an attack, we conduct an experiment from May 2019 to June 2019 (40 days). In particular, we profile 5 websites (Twitter: Houston Rockets, Trinity Church Boston, NASA Image of the Day, eBay, and Oracle Arena)[1] for the first 20 days, and observe the websites for the next 20 days to check whether the input words appear. We select an input word from each of the websites, resulting in 5 input words in total. As shown in Figure 5.1-(a), during the profiling period, there are about *19 hours that all desired input words appear together* (highlighted). Figure 5.1-(b) shows the input words that appeared on the websites during the observation period (May 28th, 2019 $\sim$ June 17th, 2019). There are about 4 hours all the input words appeared together. We present two more such experiments on our website [76], showing that

[1] https://twitter.com/houstonrockets, https://www.trinitychurchboston.org, https://www.nasa.gov/multimedia/imagegallery/iotd.html, https://www.ebay.com, https://www.theoaklandarena.com

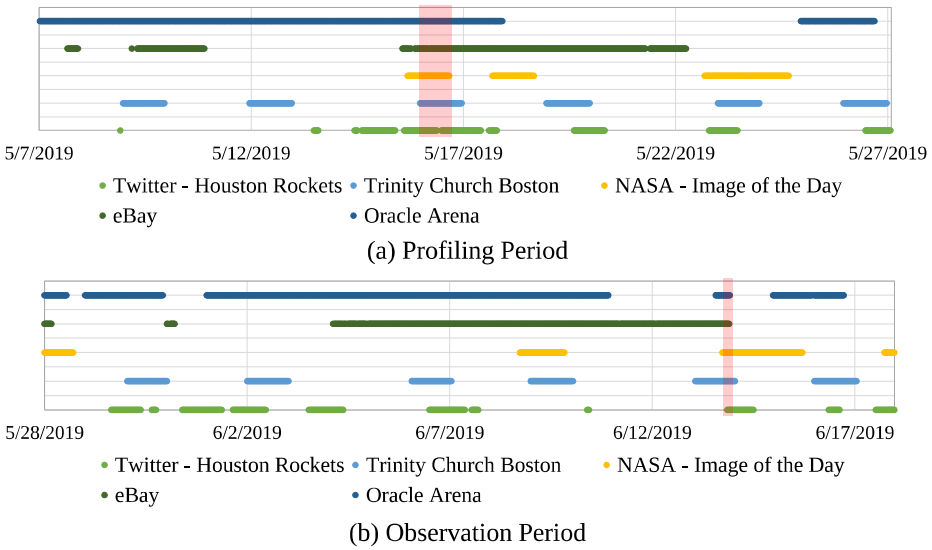


Figure 5.1: Input words appearing during the experiment. X-axis represents the date and Y-axis represent the appearance of input words from websites.

creating reliable and stealthy attacks is possible.

5.2 Anti-forensic capability of Stealthy-Attack

Datasets for payloads. We collect 573 server-side malware from known malware collection repositories [7, 8, 11, 58, 69, 79, 81, 82, 83, 85, 88]. The samples consist of eight types: webshells, backdoors, bypassers, uploaders, spammers, SQLShells, reverse shells, and flooders. For each category, we collect a similar number of samples (e.g., 61∼79). Details of each type of the samples can be found in Appendix A.

Statistics of Stealthy-Attack. We generate Stealthy-Attack instances for all 573 collected malware samples as shown in Figure 5.2. We categorize them by the samples' sizes. As shown in Figure 5.2-(a), the sizes of Stealthy-Attack are significantly larger than the original samples (from to 26 to 67 times roughly). To mitigate this significant size increase, we apply compression, e.g., gzip, to reduce the size of Stealthy-

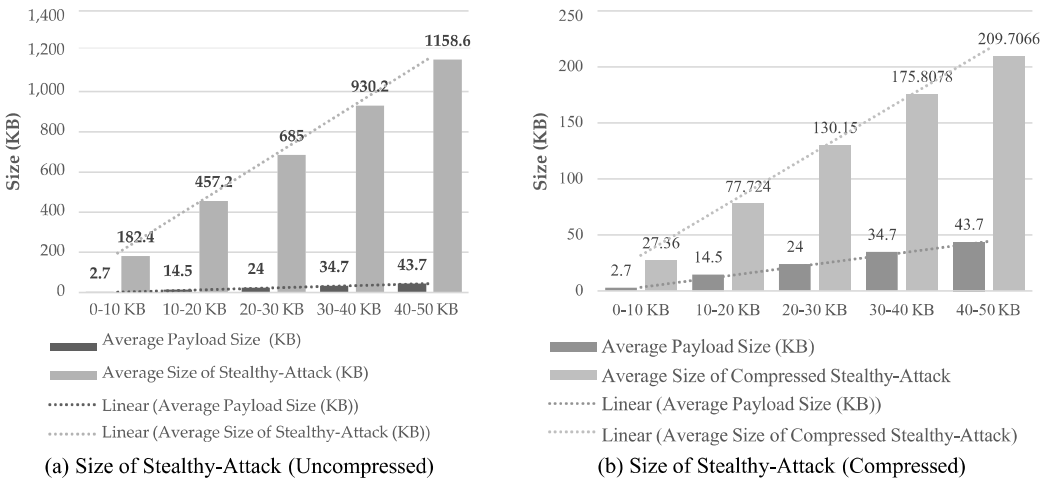


Figure 5.2: Size of Stealthy-Attack instances.

Attack. Figure 5.2-(b) shows the sizes after the compression. Except for the first group, the size of Stealthy-Attack is about 5 times larger than the original sample. Stealthy-Attack in the smallest group is 10 times bigger than the original sample. However, their sizes are less than 30 KB, which is commonly observed in real-world PHP applications.

5.3 Comparison with existing obfuscators

We compare Stealthy-Attack with state-of-the-art obfuscation techniques. We prepare two sets of samples: benign samples and malicious samples. We apply existing obfuscators to obtain obfuscated versions of samples. We also create Stealthy-Attack of the samples. Then, we run existing malware detectors to see whether the obfuscated samples and Stealthy-Attack instances are detected.

Obfuscator selection. Four state-of-the-art obfuscators are chosen based on their popularity: PHP Obfuscator [25], YAK Pro [39], Best PHP Obfuscator [10], and Simple Online PHP Obfuscator [43].

Malware detector selection. We use three widely used malware detectors (PHP Malware Finder [78], Linux Malware Detector [44], and Shellray [73]) and a recently released PHP malware scanning tool called MalMax [55] that handles multiple layers of obfuscations and exposes all hidden malicious behaviors of malware. We do not use popular anti-virus software [49, 77] because they perform worse than the malware detectors we selected as mentioned in [55].

Malicious sample selection and methodology. From the 573 malware we collected, malware samples that are *not detected by existing malware detectors* are excluded from this experiment. Specifically, PHP Malware Finder identifies 413 samples, Linux Malware Detector flags 185 samples as malware, and Shellray detects 524 samples. To this end, we use the different numbers of samples for experiments with each malware detector. Then, each obfuscator is applied to the samples and obtains obfuscated malicious payloads. Finally, the four malware detectors scan all the samples obfuscated by existing obfuscators and Stealthy-Attack instances.

Result for malicious samples. Table 5.1 shows that all existing malware detection tools are unable to detect Stealthy-Attack (0%), while most of the malware samples obfuscated by the other tools can be detected by at least three detectors: PHP Malware Finder (averagely 89.5%), Shellray (83%), and MalMax (100%). The detection rate of Linux Malware Detector (LMD) is relatively lower than others as LMD is not specifically designed for PHP server-side malware.

Understanding false alarms. Some malware detectors often consider *any obfuscated programs as malicious*, causing high false positive rates. To understand false positive, 573 benign PHP program files from popular PHP programs' codebases (including WordPress [87], Joomla [30], phpMyAdmin [22], and CakePHP [47]) are collected. Initially, none of the 573 benign files are flagged as malware by the existing

Table 5.1: Detection results on malicious and benign samples.

Obfuscator	PHP Mal. Finder		Linux Mal. Detect		Shellray		MalMax	
	Mal.	Benign	Mal.	Benign	Mal.	Benign	Mal.	Benign
PHP Obfuscator [25]	399/413	161/573	98/185	0/573	479/524	0/573	573/573	0/573
YAK Pro [39]	264/413	139/573	16/185	0/573	239/524	1/573	573/573	0/573
Best PHP Obfuscator [10]	412/413	573/573	25/185	0/573	505/524	557/573	573/573	0/573
Simple PHP Obfuscator [43]	413/413	573/573	0/185	0/573	524/524	573/573	573/573	0/573
Stealthy-Attack	0/413	0/573	0/185	0/573	0/524	0/573	0/573	0/573

Green cells on 'Mal.' columns indicate that techniques are effective against malware detectors (Detected less than 5%, and lighter green if 5%∼50%) while green cells on 'Benign' columns mean that they have no false positives (Lighter green if 5%∼50%). Red cells represent the opposite (undesirable) results.

detectors. However, once they are obfuscated by PHP Malware Finder and ShellRay, we observe many of them are detected as malware (i.e., high false positive rates) by Best PHP Obfuscator [10] and Simple Online PHP Obfuscator [43]. Linux Malware Detector has no false positives, while it misses many PHP malware samples in general (i.e., low true positive rate).

Result from MalMax. MalMax [55] detects all the malicious code hidden by the four existing obfuscators without flagging any benign obfuscated samples. However, MalMax detects none of the Stealthy-Attack instances. This is because MalMax focuses on executing all statements without precisely identifying attack triggering inputs. Simply executing all statements of a target is sufficient for analyzing the existing obfuscators but not sufficient for Stealthy-Attack.

Chapter 6

Conclusion

This research introduces a novel form of attack called Stealthy-Attack, which aims to secretly deliver malicious payloads while posing significant obstacles to forensic analysis after the attack.

We triggered attacks by leveraging popular sites that hackers cannot control. To collect words regardless of site changes, we utilized the DOM tree, and words that meet certain criteria among those collected will reappear in the near future to frustrate victims and forensic analysts from predicting the timing of the attack. Additionally, we obstruct commonly used words to prevent forensic analysts from easily guessing the attack input vector.

We leverage Fail-free Dynamic State Machine that effectively evades various forensic analysis attempts. Malicious code remains undetected until a set of appropriate benign words is translated by the state machine so that they can evade detection from the most popular detection tools. Two characteristics of state machine make the analysis attempts challenging here: Transition on any inputs and dynamic output translation. Our evaluation shows that Stealthy-Attack is highly effective in preventing forensic analysis.

Bibliography

[1] *A text file containing 479k English words*. https://github.com/dwyl/english-words. 2019.

[2] Cybersecurity & Infrastructure Security Agency. *Russian State-Sponsored Advanced Persistent Threat Actor Compromises U.S. Government Targets*. https://us-cert.cisa.gov/ncas/alerts/aa20-296a. 2020.

[3] Hyrum S Anderson et al. "Learning to evade static PE machine learning malware models via reinforcement learning". In: *arXiv preprint arXiv:1801.08917* (2018).

[4] Azeem Aqil et al. "Detection of stealthy TCP-based DoS attacks". In: *MILCOM 2015-2015 IEEE Military Communications Conference*. IEEE. 2015, pp. 348–353.

[5] Bart van Arnhem. *phpscan: Symbolic execution inspired PHP application scanner for code-path discovery*. https://github.com/bartvanarnhem/phpscan. 2017.

[6] Davide Balzarotti et al. "Saner: Composing static and dynamic analysis to validate sanitization in web applications". In: *2008 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2008, pp. 387–401.

[7] P. Bart. *PHP-backdoors: A collection of PHP backdoors*.

[8] BDLeet. *public-shell: Some Public Shell*. https://github.com/BDLeet/public-shell. 2016.

[9] Michela Becchi and Patrick Crowley. "A hybrid finite automaton for practical deep packet inspection". In: *Proceedings of the 2007 ACM CoNEXT conference*. ACM. 2007, p. 1.

[10] *Best PHP Obfuscator*. 2018. URL: http://www.pipsomania.com/best_php_obfuscator.do.

[11] BlackArch. *webshells: Various webshells*. https://github.com/BlackArch/webshells. 2019.

[12] Jerry R Burch et al. "Symbolic model checking: 1020 states and beyond". In: *Information and computation* 98.2 (1992), pp. 142–170.

[13] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs." In: *OSDI*. Vol. 8. 2008, pp. 209–224.

[14] Haibo Chen et al. "Control flow obfuscation with information flow tracking". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM. 2009, pp. 391–400.

[15] Mihai Christodorescu et al. "Semantics-aware malware detection". In: *2005 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2005, pp. 32–46.

[16] Christian Collberg, Clark Thomborson, and Douglas Low. "Manufacturing cheap, resilient, and stealthy opaque constructs". In: *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM. 1998, pp. 184–196.

[17] Johannes Dahse and Jörg Schwenk. "RIPS-A static source code analyser for vulnerabilities in PHP scripts". In: *Retrieved: February* 28 (2010), p. 2012.

[18] Derick Rethans. *Variable tracing with Xdebug — Derick Rethans*. https://derickrethans.nl/variable-tracing-with-xdebug.html. 2009.

[19] designsecurity. *progpilot: A static analysis tool for security*. https://github.com/designsecurity/progpilot. 2016.

[20] Sarang Dharmapurikar et al. "Deep packet inspection using parallel bloom filters". In: *11th Symposium on High Performance Interconnects, 2003. Proceedings*. IEEE. 2003, pp. 44–51.

[21] László Erdődi and Audun Jøsang. "Exploitation vs. Prevention: The Ongoing Saga of Software Vulnerabilities". In: *Acta Polytechnica Hungarica* 17.7 (2020).

[22] Maurício Meneghini Fauth. *phpmyadmin: A web interface for MySQL and MariaDB*. https://github.com/phpmyadmin/phpmyadmin. 2019.

[23] Daniele Filaretti and Sergio Maffeis. "An executable formal semantics of PHP". In: *European Conference on Object-Oriented Programming*. Springer. 2014.

[24] FIREEYE. *APT41: Double Dragon, a dual espionage and cyber crime operation*. https://content.fireeye.com/apt-41/rpt-apt41. 2019.

[25] Maurice Fonk. *php-obfuscator: A parsing PHP obfuscator*. https://github.com/naneau/php-obfuscator. 2019.

[26] Yanick Fratantonio et al. "Triggerscope: Towards detecting logic bombs in android applications". In: *2016 IEEE symposium on security and privacy (SP)*. IEEE. 2016, pp. 377–396.

[27] Heilan Yvette Grimes. "Eir - Static Vulnerability Detection in PHP Applications". In: (2015).

[28] David Hauzar and Jan Kofroň. "WeVerca: Web Applications Verification for PHP". In: *International Conference on Software Engineering and Formal Methods*. Springer. 2014, pp. 296–301.

[29] Torben Jensen et al. "Thaps: automated vulnerability scanning of php applications". In: *Nordic conference on secure IT systems*. Springer. 2012, pp. 31–46.

[30] *Joomla: Content Management System (CMS)*. https://www.joomla.org/. 2019.

[31] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Pixy: A static analysis tool for detecting web application vulnerabilities". In: *2006 IEEE Symposium on Security and Privacy (S&P)*. IEEE. 2006, 6–pp.

[32] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. "Static analysis for detecting taint-style vulnerabilities in web applications". In: *Journal of Computer Security* 18.5 (2010), pp. 861–907.

[33] Chijung Jung et al. "Defeating Program Analysis Techniques via Ambiguous Translation". In: *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2021, pp. 1382–1387.

[34] Chijung Jung et al. "Hiding Critical Program Components via Ambiguous Translations". In: *2022 IEEE/ACM 44rd International Conference on Software Engineering (ICSE)*. IEEE. 2022.

[35] Alexandros Kapravelos et al. "Revolver: An automated approach to the detection of evasive web-based malware". In: *Presented as part of the 22nd USENIX Security Symposium*. 2013, pp. 637–652.

[36] Ranjita Pai Kasturi et al. "TARDIS: Rolling Back The Clock On CMS-Targeting Cyber Attacks". In: *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020, pp. 1156–1171. DOI: 10.1109/SP40000.2020.00116. URL: https://doi.org/10.1109/SP40000.2020.00116.

[37] Kyungtae Kim et al. "J-force: Forced execution on javascript". In: *Proceedings of the 26th international conference on World Wide Web*. International World Wide Web Conferences Steering Committee. 2017, pp. 897–906.

[38] Johannes Kinder et al. "Detecting malicious code by model checking". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2005, pp. 174–187.

[39] Pascal Kissian. *YAK Pro: Php Obfuscator*. https://www.php-obfuscator.com/. 2019.

[40] Etienne Kneuss, Philippe Suter, and Viktor Kuncak. "Phantm: PHP analyzer for type mismatch". In: *FSE'10 Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. CONF. 2010.

[41] Bojan Kolosnjaji et al. "Adversarial malware binaries: Evading deep learning for malware detection in executables". In: *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE. 2018, pp. 533–537.

[42] Sailesh Kumar et al. "Algorithms to accelerate multiple regular expressions matching for deep packet inspection". In: *ACM SIGCOMM Computer Communication Review*. Vol. 36. 4. ACM. 2006, pp. 339–350.

[43] Robert Lie. *Simple online PHP obfuscator: encodes PHP code into random letters, numbers and/or characters*. https://www.mobilefish.com/services/php_obfuscator/php_obfuscator.php. 2019.

[44] *Linux Malware Detect*. https://www.rfxn.com/projects/linux-malware-detect/. 2019.

[45] CPO Magazine. *New Report Reveals Chinese APT Groups May Have Been Entrenched in Some Servers for Nearly a Decade Using Little-Known Linux Exploits, CPO Magazine*. https://www.cpomagazine.com/cyber-security/new-report-reveals-chinese-apt-groups-may-have-been-entrenched-in-some-servers-for-nearly-a-decade-using-little-known-linux-exploits/. 2020.

[46] Jian Mao et al. "Detecting malicious behaviors in javascript applications". In: *IEEE Access* 6 (2018), pp. 12284–12294.

[47] Larry Masters. *CakePHP: The Rapid Development Framework for PHP*. https://cakephp.org/. 2019.

[48] Ibéria Medeiros, Nuno F Neves, and Miguel Correia. "Automatic detection and correction of web application vulnerabilities using data mining to predict false positives". In: *Proceedings of the 23rd international conference on World wide web*. ACM. 2014, pp. 63–74.

[49] Microsoft. *Microsoft Defender Advanced Threat Protection*. https://docs.microsoft.com/en-us/windows/security/threat-protection/microsoft-defender-atp/microsoft-defender-advanced-threat-protection. 2019.

[50] Microsoft. *Z3Prover/z3: The Z3 Theorem Prover*. https://github.com/Z3Prover/z3. 2020.

[51] Ondřej Mirtes. *phpstan: PHP Static Analysis Tool*. https://github.com/phpstan/phpstan. 2019.

[52] MITRE. *NVD - CVE-2020-7048*. https://nvd.nist.gov/vuln/detail/CVE-2020-7048. 2020.

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Exploring multiple execution paths for malware analysis". In: *2007 IEEE Symposium on Security and Privacy*. IEEE. 2007, pp. 231–245.

[54] Mozilla. *WebSocket - Web APIs | MDN*. https://developer.mozilla.org/en-US/docs/Web/API/WebSocket. 2020.

[55] Abbas Naderi-Afooshteh et al. "MalMax: Multi-Aspect Execution for Automated Dynamic Web Server Malware Analysis". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 1849–1866.

[56] Paco Nathan. *PyTextRank, a Python implementation of TextRank for text document NLP parsing and summarization*. https://github.com/ceteri/pytextrank/. 2016.

[57] Hung Viet Nguyen et al. "Auto-locating and fix-propagating for HTML validation errors to PHP server-side code". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society. 2011, pp. 13–22.

[58] nixawk. *fuzzdb: Web Fuzzing Discovery and Attack Pattern Database*. https://github.com/nixawk/fuzzdb. 2018.

[59] *NPR: National Public Radio*. https://npr.org/. 2019.

[60] *NPR: News and National Top Stories*. https://npr.org/sections/national/. 2019.

[61] Paulo Jorge Costa Nunes, José Fonseca, and Marco Vieira. "phpSAFE: A security analysis tool for OOP web application plugins". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015.

[62] Oswaldo Olivo. *TaintPHP: Static Taint Analysis for PHP web applications*. https://github.com/olivo/TaintPHP. 2016.

[63] OneSourceCat. *phpvulhunter: A tool that can scan php vulnerabilities automatically using static analysis methods*. https://github.com/OneSourceCat/phpvulhunter. 2015.

[64] Ioannis Papagiannis, Matteo Migliavacca, and Peter Pietzuch. "PHP Aspis: using partial taint tracking to protect against injection attacks". In: *2nd USENIX Conference on Web Application Development*. Vol. 13. 2011.

[65] Fei Peng et al. "X-force: force-executing binary programs for security applications". In: *23rd USENIX Security Symposium*. 2014, pp. 829–844.

[66] *PHP: Pspell Functions*. https://www.php.net/manual/en/ref.pspell.php. 2019.

[67] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. "Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat". In: *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, pp. 68–78.

[68] Mila Dalla Preda et al. "A semantics-based approach to malware detection". In: *ACM SIGPLAN Notices* 42.1 (2007), pp. 377–388.

[69] Ridter. *Pentest*. https://github.com/Ridter/Pentest. 2019.

[70] Dewhurst Ryan. "Implementing basic static code analysis into integrated development environments (ides) to reduce software vulnerablitilies". In: *A Report submitted in partial fulfillment of the regulations governing the award of the Degree of BSc (Honours) Ethical Hacking for Computer Security at the University of Northumbria at Newcastle* 2012 (2011).

[71] Prateek Saxena et al. "A symbolic execution framework for javascript". In: *2010 IEEE Symposium on Security and Privacy*. IEEE. 2010, pp. 513–528.

[72] Sebastian Schrittwieser et al. "Covert computation: Hiding code in code for obfuscation purposes". In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*. ACM. 2013, pp. 529–534.

[73] *Shellray: A PHP webshell detector*. https://shellray.com/. 2019.

[74] Justine Sherry et al. "Blindbox: Deep packet inspection over encrypted traffic". In: *ACM SIGCOMM Computer communication review* 45.4 (2015), pp. 213–226.

[75] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. "Unearthing stealthy program attacks buried in extremely long execution paths". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2015, pp. 401–413.

[76] *Stealthy-Attack: Supplementary Materials*. https://sites.google.com/virginia.edu/stealthy-attack-supplementary/home. 2023.

[77] Symantec. *Norton™ - Antivirus & Anti-Malware Software*. https://us.norton.com/. 2019.

[78] NBS Systems. *GitHub - nbs-system/php-malware-finder: Detect potentially malicious PHP files*. https://github.com/nbs-system/php-malware-finder/. 2019.

[79] tanjiti. *webshellSample: Webshell sample for WebShell Log Analysis*. https://github.com/tanjiti/webshellSample. 2018.

[80] Teryl Taylor et al. "Detecting malicious exploit kits using tree-based similarity searches". In: *proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM. 2016, pp. 255–266.

[81] tennc. *webshell: A webshell open source project*. https://github.com/tennc/webshell. 2019.

[82] John Troon. *php-webshells: Common php webshells*. https://github.com/JohnTroony/php-webshells. 2016.

[83] tutorial0. *WebShell: WebShell Collect*. https://github.com/tdifg/WebShell. 2016.

[84] vimeo. *psalm: A static analysis tool for finding errors in PHP applications*. https://github.com/vimeo/psalm. 2019.

[85] *VirusShare*. https://virusshare.com/. 2019.

[86] David Wagner and R Dean. "Intrusion detection via static analysis". In: *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P*. IEEE. 2000, pp. 156–168.

[87] *WordPress*. https://wordpress.com/. 2019.

[88] xl7dev. *WebShell: Webshell & Backdoor Collection*. https://github.com/xl7dev/WebShell. 2017.

[89] Quan Yang. *Taint-em-All: A taint analysis tool for the PHP language*. https://github.com/quanyang/Taint-em-All. 2019.

Appendices

Appendix A

Payload types

A webshell is malware that enables attackers to access a compromised server via a web browser that acts like a command-line interface. Backdoor is used to provide remote access to an infected machine for attackers. Bypassers are used to avoid detections of local or remote security mechanisms (e.g., firewalls). Uploaders are used to remotely inject additional malware into victim machines. Spammers compose and send spoof/spam emails. SQLShells allows remote attackers to access databases of compromised servers, similar to webshells. A reverse shell is a type of shell that communicates back to the attacker's machine from a victim's machine. Flooders are used to launch Denial of Service (DoS) attacks by sending an excessive number of network packets.