electronics

Article

Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability

Jiazhen Zhao [1,2], Yuliang Lu [1,2,*], Kailong Zhu [1,2], Zehan Chen [1,2] and Hui Huang [1,2]

1 College of Electronic Engineering, National University of Defense Technology, Hefei 230037, China; jiazhenzhao@nudt.edu.cn (J.Z.); zhukailong@nudt.edu.cn (K.Z.); zehan_chen086@nudt.edu.cn (Z.C.); huanghui17@nudt.edu.cn (H.H.)
2 Anhui Province Key Laboratory of Cyberspace Security Situation Awareness and Evaluation, Hefei 230037, China
* Correspondence: luyuliang@nudt.edu.cn

Abstract: Current static detection technology for web application vulnerabilities relies highly on specific vulnerability patterns, while dynamic analysis technology has the problem of low vulnerability coverage. In order to improve the ability to detect unknown web application vulnerabilities, this paper proposes a PHP Remote Command/Code Execution (RCE) vulnerability directed fuzzing method. Our method is a combination of static and dynamic methods. First, we obtained the potential RCE vulnerability information of the web application through fine-grained static taint analysis. Then we performed instrumentation for the source code of the web application based on the potential RCE vulnerability information to provide feedback information for fuzzing. Finally, a loop feedback web application vulnerability automatic verification mechanism was established in which the vulnerability verification component provides feedback information, and the seed mutation component improves the vulnerability test seed based on the feedback information. On the basis of this method, the prototype system Cefuzz (Command/Code Execution Fuzzer) is implemented. Thorough experiments show that, compared with the existing web application vulnerability detection methods, Cefuzz significantly improves the verification effect of RCE vulnerabilities, discovering 13 unknown vulnerabilities in 10 popular web CMSes.

Keywords: RCE vulnerability; taint analysis; directed fuzzing; vulnerabilities verification; Cefuzz

Citation: Zhao, J.; Lu, Y.; Zhu, K.; Chen, Z.; Huang, H. Cefuzz: An Directed Fuzzing Framework for PHP RCE Vulnerability. *Electronics* **2022**, *11*, 758. https://doi.org/10.3390/electronics11050758

Academic Editors: Stavros Shiaeles, Bogdan Ghita and Nicholas Kolokotronis

Received: 22 December 2021
Accepted: 26 February 2022
Published: 1 March 2022

1. Introduction

Web applications are the primary way the Internet provides information and services today. As of August 2021, the number of global websites has reached 1,211,444,849 [1]. Covering government digital services, social media, shopping websites, online education, online banking, remote office, etc., web services [2] have already become an indispensable part of people's lives. Once a serious vulnerability occurs in a popular web application, it will not only affect the normal daily use of many netizens, but also involve serious user privacy issues. To analyze and detect web vulnerabilities, and to repair them before a Cybersecurity incident occurs, is an important means to ensure cyberspace security. There are various types of vulnerabilities in web applications, among which RCE vulnerabilities, namely Remote Command Execution [3] and Remote Code Execution [4], are the most common and serious. Empirically, attackers can execute system commands or arbitrary codes on the target system using the above two types of vulnerabilities to obtain system authority, destroy the integrity and usability of the target system, and steal user privacy information. According to statistics from W3tech, 78.9% of websites use PHP as their back-end development language until 11 September 2021 [5]. Due to the widespread use of the PHP language in web applications, there are a large number of security vulnerabilities in PHP web applications. To this end, this paper mainly studies the detection methods of PHP RCE vulnerabilities.

Currently, web application vulnerabilities detection technology can be roughly divided into static analysis technology and dynamic analysis technology.

The static analysis technology does not consider the runtime state of the web applications [6,7]. By analyzing the code structure and functions in the source code that do not conform to the security rules, the security flaws that may be hidden in the web applications are discovered. Therefore, static detection with a high detection coverage rate and a low false-negative rate can find considerable amounts of potential vulnerabilities. However, its shortcomings are also very obvious. Due to the ignorance of the runtime state of web applications, most of the potential vulnerabilities discovered by static methods are not reachable and not exploitable, and the detection false positive rate is high.

There are two types of dynamic analysis techniques, black box dynamic analysis and white box dynamic analysis. The black box dynamic analysis communicates with the web applications through the web front end, identifying potential vulnerabilities in the web applications by mining access paths as many as possible based on a web crawler [8]. White box dynamic analysis monitors program execution through plug-ins and kernel hooks, and alerts when risky events occur, just as unsafe function calls are recognized [9]. The dynamic analysis considers the runtime state of the web applications, so the detected vulnerabilities are reachable. However, its defects are also very obvious. As dynamic analysis cannot scan the source code of a web application, it is difficult for analysis to cover deep code, so there is a high false negative rate and it is affected by the uncertainty introduced at runtime.

In recent decades, the role of taint analysis in program analysis has received extensive attention from researchers. Static taint analysis refers to the detection of whether data can propagate from the source of the taint to the sink point by analyzing the data dependency between program variables without running and modifying the code [10,11]. The object of static taint analysis is generally the source code or intermediate representation of the program.

The basic idea of fuzzing [12] is to randomly generate a large number of test cases, drive the target program to run, monitor the running state of the program, and find vulnerabilities based on whether an exception occurs. Due to the effectiveness, practicality, and vulnerability discovery capabilities of the gray box fuzzing [13], it has become very popular in the field of binary vulnerability mining. However, the programming languages of web applications are varied, and it is difficult to conduct unified research on the basis of multiple programming languages, so there is currently limited research using gray box fuzzing in the field of web application vulnerability mining. Among them, however, Navex [14] is one of the well-known works. Navex [14] utilizes static parsing to perform mapping between source variables, such as URL parameters, which are server-side code statements (such as print commands) that output the source statements back to the client. Creating this source-sink link and determining whether cleanup occurs during this process is a key factor in exposing the vulnerability. However, there is an expensive constraint solving operation for each associated pair to be created, and the severity of this problem only increases with the number of sources and sinks.

Fuzzing has the advantages of fast test case generation, low overhead, and strong scalability. However, due to the use of random mutation methods, fuzzing is difficult to bypass strict constraints and has low code coverage. For various types of vulnerabilities in web applications, a payload that can successfully trigger a vulnerability often requires a certain vulnerability grammatical structure. Applying fuzzing to test vulnerabilities in web applications will cause the problem that random test cases do not make sense for discovering webapp vulnerabilities. In addition, most of the current fuzzing methods applied to web applications only use the web application response page information to guide the seed selection and mutation. The fuzzer only knows whether the vulnerability is triggered, and cannot know how the depth of each test case can reach in programs, which means the insufficient amount of feedback information. For this reason, the fuzzer cannot know from this round of testing which seeds are better, so it essentially becomes a brute force traversal tool.

In response to the above challenges, this paper proposes a PHP RCE vulnerability detection algorithm based on guided fuzzing. The main contributions of this paper are as follows:

(1) Guided by potential RCE vulnerability information from static taint analysis, we perform instrumentation for the source code of the web application to improve the perceptual ability of program status during fuzzing.
(2) We use three types of special initial seeds that easily show command execution results, and propose six seed mutation operations, combined with three vulnerability monitoring methods, which can accurately capture the signal that the vulnerabilities are successfully verified.
(3) A PHP RCE vulnerability directed fuzzing prototype system (Cefuzz) is implemented. Experimental results show that Cefuzz detects 13 unknown vulnerabilities in 10 popular web CMSes, which demonstrates Cefuzz has strong power in discovering unknown vulnerabilities compared with the current popular web vulnerability detection methods.

The remainder of this paper is organized as follows. Section 2 introduces the limitations of current fuzzing technology in web application vulnerability analysis. The overview of the proposed approach is described in Section 3. Section 4 describes the details of key steps in our method. Section 5 evaluates our method. We summarize the related work in Section 6 and provide our conclusions in Section 7.

2. Motivation

At present, there are many static analysis tools for PHP web applications, but we know that the results obtained by static analysis are not necessarily accurate, and manual verification of the analysis results is required, so our starting point is to take the results of static analysis as the guide, conduct guided fuzzing, and realize the automatic verification of vulnerabilities.

In the fuzzing workflow, seed selection and mutation are important parts. According to the feedback information [15] from the target program, seed selection and mutation are carried out, new test cases are generated, and the next round of testing is carried out. The quality of test cases determines the analysis effect of this round of testing. At the same time, the quality of seed selection and mutation determines the quality of the generated test. Furthermore, the information feedback by the target program is the key information that determines the rules of seed selection and mutation. The higher the amount of test feedback information, the more dimensions or more options for seed selection can be made. Information determines which mutation strategy to use.

According to the method of generating test cases, the fuzzing technology can be divided into completely random fuzzing [16], mutation-based fuzzing [17], and generation-based fuzzing [18]. Among the above technologies, random fuzzing is difficult to meet the HTTP protocol [19] specification; the mutation-based fuzzing requires a long mutation time to generate a seed for HTTP protocol; the generation-based fuzzing can quickly generate suitable test cases. However, web generation-based fuzzing experiences a bit of feedback information from the feedback module, which makes it difficult to mutate excellent seeds. The following will introduce the problems of the current three technologies in detail and introduce how our methods solve the current problems.

Dynamic vulnerability analysis of web applications requires interaction with web applications based on the HTTP protocol. The random-test fuzzing cases generate lots of random strings as test cases. As the HTTP protocol specification is not met, the normal response of the web application cannot be obtained, and this technology is difficult to apply in the web application vulnerability analysis, as shown in the Figure 1.

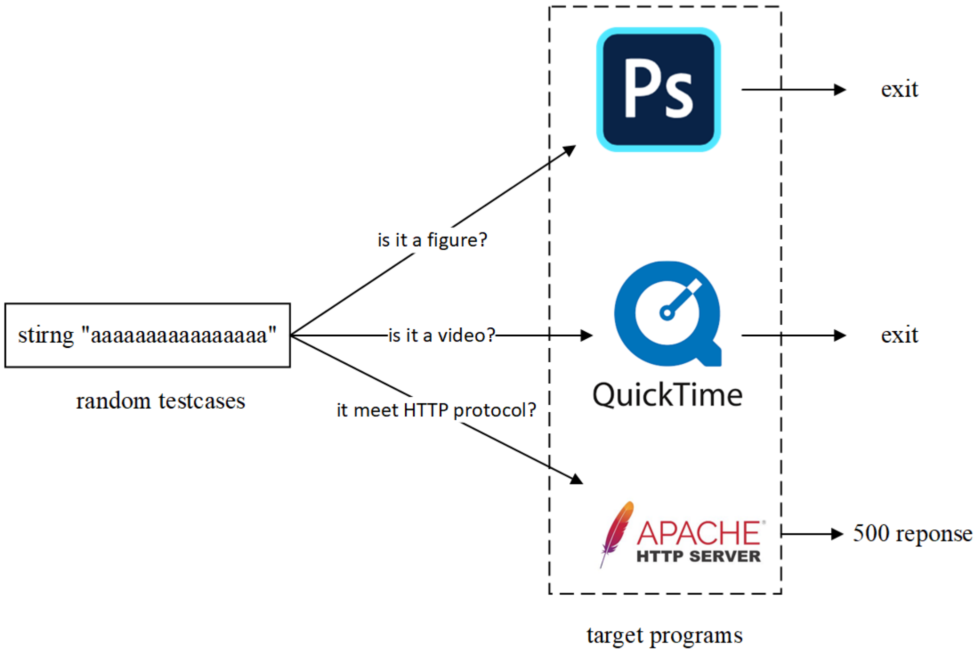


Figure 1. Completely randomly generated test cases cannot pass format verification.

The mutation-based fuzzing introduces the concept of seeds. Researchers select a batch of initial seeds and then mutate the initial seeds through mutation operations such as replacement, insertion, and deletion to create test cases. The test cases are scored based on code coverage and path coverage. Next, the seeds with excellent performance are selected for the next round of mutation. Using mutation-based fuzzing to analyze web application vulnerabilities, due to the random selection of mutation rules, it is likely that the seeds generated by mutation do not meet the HTTP protocol specifications. However, the existence of the seed selection mechanism makes it generally possible for the seed that meets the HTTP protocol specification to stay after multiple rounds of mutation selection, but its required a lot of mutation time.

Generation-based fuzzing is the most used fuzzing technology in the field of web application vulnerability analysis. Its test cases are based on the grammatical rules and protocol format set by the researcher, and the input is generated based on the model. The test cases generally meet the HTTP protocol specifications and can achieve normal responses from web applications.

However, currently, generation-based fuzzing technologies are mainly based on web application response results as feedback information, combined with some indirect information, such as page similarity (the similarity of the normal request–response page and the vulnerability test response page), the server response time (long response time represents may trigger vulnerabilities), etc. Such indirect information can only determine whether the seed triggers web application vulnerabilities, but cannot know the reason why the vulnerabilities cannot be triggered, nor can it know the depth of execution of the seed in the web application, and the feedback information is too limited. This caused a serious problem. We cannot know why the current test seed cannot trigger the vulnerability, nor can we know whether the test seed has reached the target vulnerability area. This makes it impossible to select the seeds that performed well in the previous round in the next round of testing, resulting in an ineffective mutation feedback mechanism.

Our method uses the vulnerable code relevant information (potential RCE vulnerability information) provided by static analysis as a lighthouse. The vulnerable code-relevant information includes the file path that may have vulnerabilities and potentially dangerous code lines, etc. Then, we perform checkpoint [20] instrumentation on the target web application source code and insert output statements in the checkpoint context which can greatly increase the amount of feedback information during fuzzing. The checkpoint here means the branch node from the program entry to the target code line. The comparison between our method and the existing method is shown in Figure 2. The existing methods (a) can only determine whether the vulnerability is successfully triggered based on the response information of the web application. The internal state of the target program is in an unknown state, which is essentially a black box test. Our method (b) can learn the passing of each checkpoint. According to the checkpoint passing situation, the test seeds with better performance are selected for the next round of mutation. In addition, we use a preset vulnerability payload (such as printing special keywords, etc.) which makes it possible to know whether the vulnerability is successfully triggered through the response information of the web application.

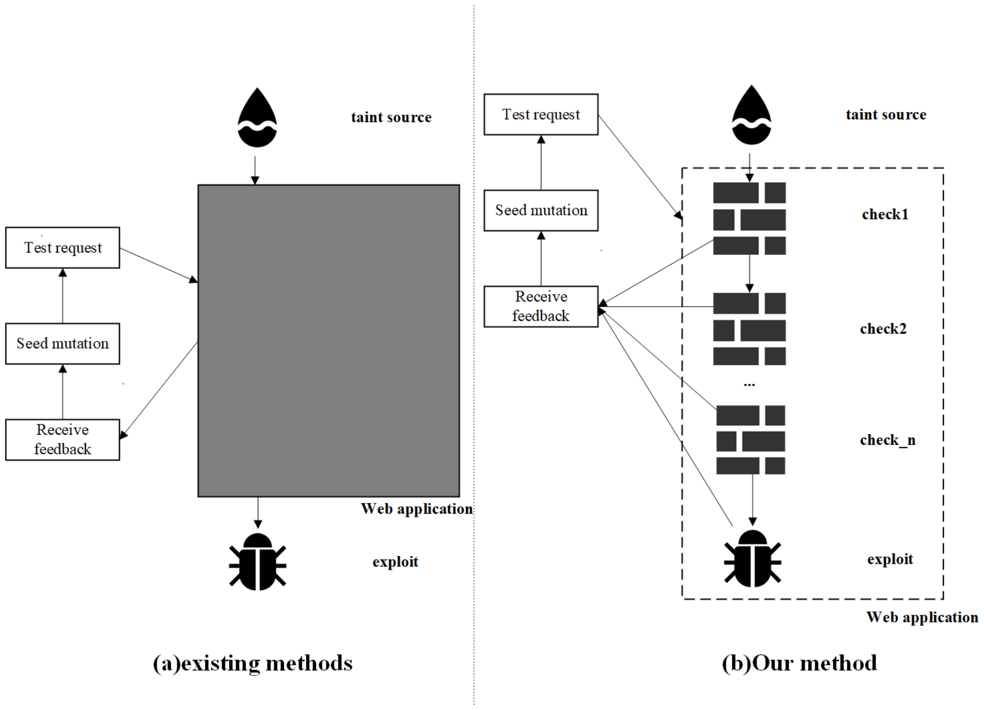


Figure 2. Comparison of our method with existing methods.

3. Overview

To solve the above limitations in the existing methods, we propose an automatic verification method for PHP RCE vulnerabilities based on directed fuzzing. This method aims to improve the ability of automatic verification of PHP RCE vulnerabilities.

The method proposed in this paper includes the following six steps, as shown in Figure 3. First, the source code analysis of the target web application is carried out by static taint analysis to obtain the code area information of the possible vulnerability. Using this information as a guide, we performed checkpoint instrumentation on the web application source code. Some status output statements are inserted so that the current execution path depth can be obtained during the fuzzing. Therefore, the fuzzer can select testing seed based on the current execution path depth, ultimately enabling the testing to reach the possibly vulnerable code area, increasing the likelihood of triggering the vulnerability.

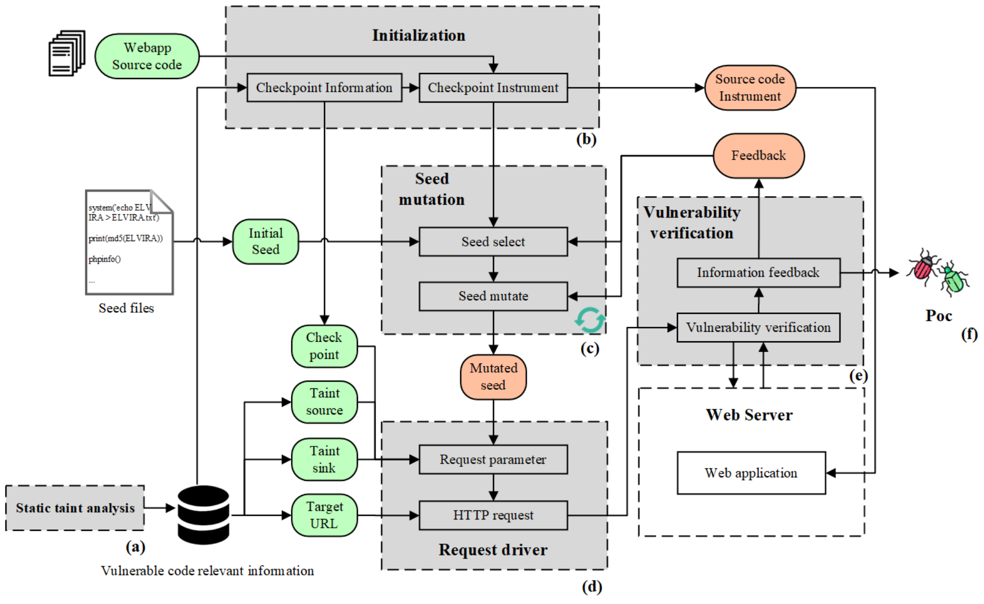


Figure 3. The overview of our method.

Static taint analysis (a) performed on the web application source code, and output related information about potential RCE vulnerabilities (taint source points, taint propagation process, and taint sink points). This part converts the web application source code into PHP intermediate representation opcodes and conducts taint analysis on opcodes. The semantics of opcodes are accurate. We have built a complete vulnerability analysis framework based on opcodes to acquire more precise analysis results.

Initialization (b) preprocesses the vulnerable code relevant information provided by static taint analysis, and obtains web application source code and seed files.

Seed mutation (c) completes the seed selection and seed mutation operation, which are key steps in fuzzing. According to the feedback information of the vulnerability verification (e), the seed selection and seed mutation are carried out according to the corresponding algorithm.

Request driver (d) generates an HTTP request to target web applications based on the seed provided by the seed mutation, the target URL, and vulnerable code relevant information generated during initialization (b).

Vulnerability verification (e) receives the response information from the web server to determine whether the vulnerability is successfully triggered.

If it is not triggered successfully, the checkpoint pass situations are fed back to the seed mutation;

If verification of vulnerability is successful, the HTTP request packet is extracted by the **vulnerability verification code output** (f), replacing the corresponding part in the vulnerability verification code template file to generate the PoC.

Each step will be described in detail in the next section.

4. Methodology

This section uses Listing 1 as an example to show the details of each step. In sample code 1, an attacker submits data through $_REQUEST ['ip'] and meets the checkpoint requirements, the submitted data are transmitted to the shell_exec function and executed, forming an RCE vulnerability.

Listing 1. Sample code 1.

```
<?php
if( isset( $_POST[ 'Submit' ]  ) ) {
// Get input
$target = $_REQUEST[ 'ip' ];
// check point1
if( strstr( $target, " " )  ){
exit("Find a space");
}
// check point 2
if( strstr( $target, "&&" )  ){
exit("Find a &&");
}
// check point 3
if( !isset( $_GET[ 'pass' ] )  ){
exit("GET_pass not set");
}
// Determine OS and execute the ping command.
if( stristr( php_uname( 's' ), 'Windows NT' ) ) {
// Windows
$cmd = shell_exec( 'ping  ' . $target );
}
else {
// *nix
$cmd = shell_exec( 'ping  -c 4 ' . $target );
}
// Feedback for the end user
$html .= "<pre>{$cmd}</pre>";
}
?>
```

4.1. Static Taint Analysis

Precise information on potential vulnerabilities provides the direction of mutation for fuzzing, which is a necessary prerequisite for the successful exploit of vulnerabilities and verification of vulnerabilities in the dynamic verification. The framework of the static taint analysis module in this paper is shown in Figure 4. This module uses our previous research achievement WTA [21], which is a prototype system that can perform fine-grained taint analysis on opcodes. On the basis of WTA, we expanded and improved the dangerous function part of the auxiliary data to meet the analysis requirements of RCE vulnerabilities, and modified the result output module to output information related to vulnerable code.

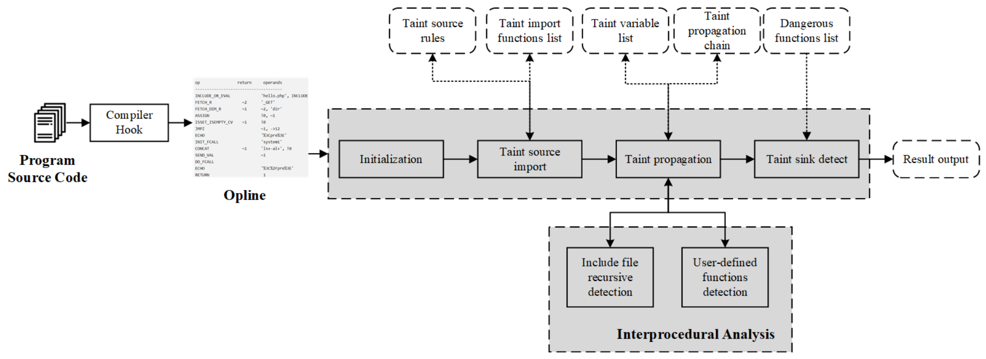
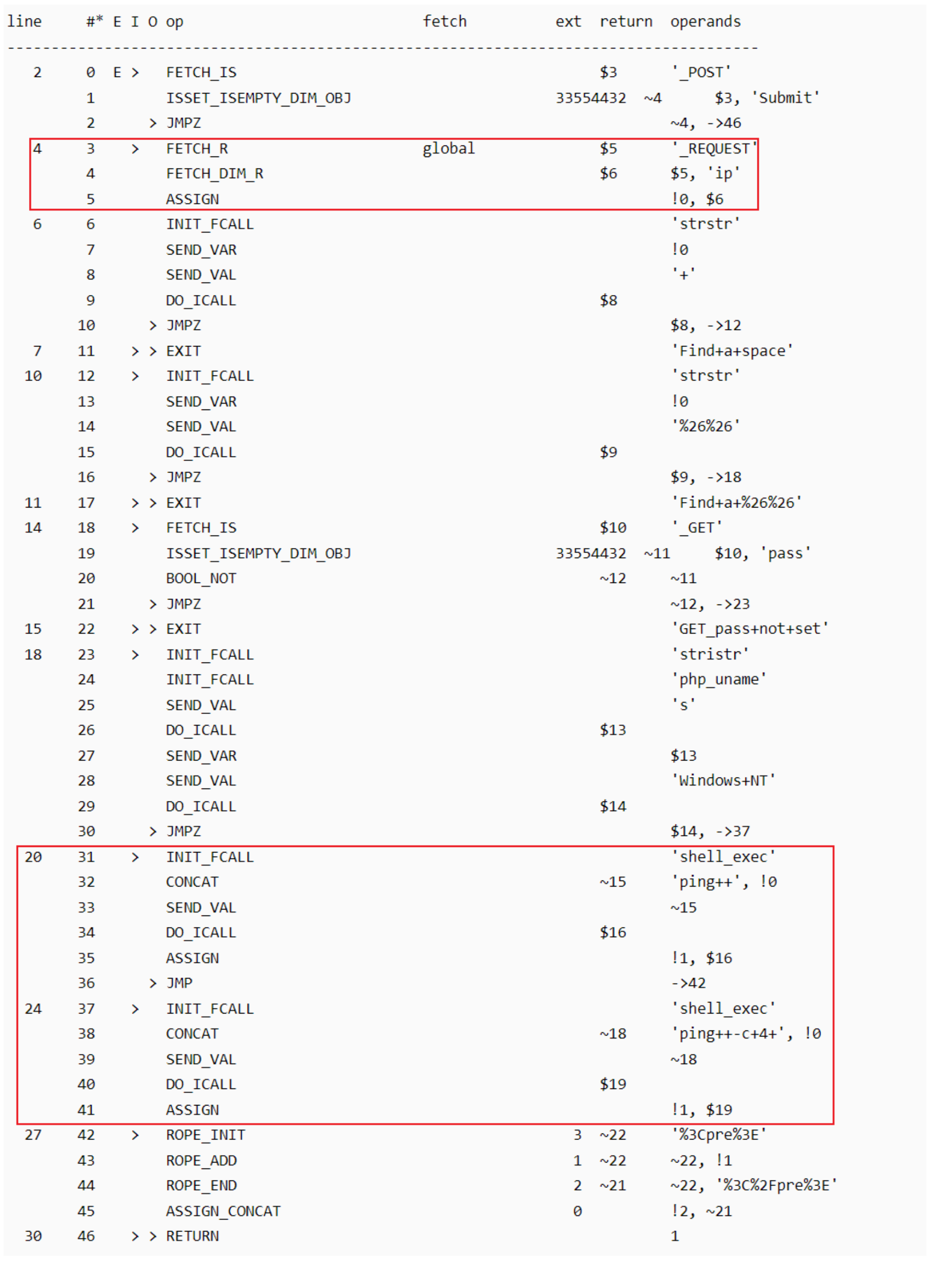


Figure 4. The overview of static taint analysis in our method.

Static taint analysis first converts the source code to IR opcodes through the compiler hook; the auxiliary data structure cooperates with the taint analysis component to perform taint analysis on the opcodes; and the interprocedural analysis component is used in the taint propagation analysis process; finally, the information about the vulnerable code is output. This information includes the path of the file where the vulnerable code is located, the program control flow graph of the file where the vulnerable code is located, and the taint propagation chain (including taint source point, taint propagation, and taint sink points).

Opcodes converted from sample code 1 is shown in Figure 5, where the taint source point is line 4 in Figure 5 (introducing the variable IP controllable by hackers), taint propagation refers to the propagation process of the tainted variable in Figure 5 $5-> $6->!0->shell_exec, and taint sink points are line 20 and line 24 in Figure 5 (the tainted variable is passed in and executes the dangerous function).



```
line     #* E I O op                           fetch          ext  return  operands
-------------------------------------------------------------------------------------
   2     0  E >   FETCH_IS                                          $3      '_POST'
         1        ISSET_ISEMPTY_DIM_OBJ                       33554432  ~4      $3, 'Submit'
         2      > JMPZ                                                      ~4, ->46
4        3    >   FETCH_R                      global               $5      '_REQUEST'
         4        FETCH_DIM_R                                       $6      $5, 'ip'
         5        ASSIGN                                                    !0, $6
   6     6        INIT_FCALL                                                'strstr'
         7        SEND_VAR                                                  !0
         8        SEND_VAL                                                  '+'
         9        DO_ICALL                                          $8
        10      > JMPZ                                                      $8, ->12
   7    11    > > EXIT                                                      'Find+a+space'
  10    12    >   INIT_FCALL                                                'strstr'
        13        SEND_VAR                                                  !0
        14        SEND_VAL                                                  '%26%26'
        15        DO_ICALL                                          $9
        16      > JMPZ                                                      $9, ->18
  11    17    > > EXIT                                                      'Find+a+%26%26'
  14    18    >   FETCH_IS                                          $10     '_GET'
        19        ISSET_ISEMPTY_DIM_OBJ                       33554432  ~11     $10, 'pass'
        20        BOOL_NOT                                          ~12     ~11
        21      > JMPZ                                                      ~12, ->23
  15    22    > > EXIT                                                      'GET_pass+not+set'
  18    23    >   INIT_FCALL                                                'stristr'
        24        INIT_FCALL                                                'php_uname'
        25        SEND_VAL                                                  's'
        26        DO_ICALL                                          $13
        27        SEND_VAR                                                  $13
        28        SEND_VAL                                                  'Windows+NT'
        29        DO_ICALL                                          $14
        30      > JMPZ                                                      $14, ->37
  20    31    >   INIT_FCALL                                                'shell_exec'
        32        CONCAT                                            ~15     'ping++', !0
        33        SEND_VAL                                                  ~15
        34        DO_ICALL                                          $16
        35        ASSIGN                                                    !1, $16
        36      > JMP                                                       ->42
  24    37    >   INIT_FCALL                                                'shell_exec'
        38        CONCAT                                            ~18     'ping++-c+4+', !0
        39        SEND_VAL                                                  ~18
        40        DO_ICALL                                          $19
        41        ASSIGN                                                    !1, $19
  27    42    >   ROPE_INIT                                      3  ~22     '%3Cpre%3E'
        43        ROPE_ADD                                       1  ~22     ~22, !1
        44        ROPE_END                                       2  ~21     ~22, '%3C%2Fpre%3E'
        45        ASSIGN_CONCAT                                  0          !2, ~21
  30    46    > > RETURN                                                    1
```

Figure 5. Opcodes converted form sample code 1.

It is worth noting that we hope that the fuzzing can be tested towards the static analysis resulting area, that is, the results obtained by the static analysis guide the direction of variation of the fuzzing.

4.2. Initialization

The three main tasks in initialization are: (1) Prepare request parameters and target URL for the **request driver**; (2) Prepare initial seeds for **seed mutation**; and (3) Extract checkpoints to perform **checkpoint instrumentation** on the web application source code, as shown in Figure 3 (The flow of the green nodes).

Request driver is responsible for generating the HTTP request packet, which requires the HTTP request method (GET/POST), the corresponding request parameters, and the request URL. This information is provided by static analysis results.

Seed mutation requires initial seed generation test cases, and the content in the seed file is initially extracted to provide initial seeds for seed mutation.

Checkpoint instrumentation is important to increase the amount of fuzzing feedback information in this framework. By inserting specific string output sentences in the checkpoint context, it can provide the seed mutation module with test case execution depth and improve the effect of seed selection and mutation. The initialization is based on the taint source point and the taint sink point, and the control flow analysis of the program control flow graph of the PHP code file is performed to find the control flow branch (checkpoint) encountered during the taint propagation process. After calculating the line number of the checkpoint in the source code according to Algorithm 1, we used Algorithm 2 to instrument the source code.

Algorithm 1 Checkpoint Found

```
Input: CFG of potentially vulnerable file: $CFG$, Sink point located line No. : $target\_line$
Output: Check point located line No. : $check\_line$
 1: target_node = node_search($CFG$, $target\_line$)
 2: reverse_nodes = BFS_predecessors_searcch($CFG$,target_node) // BFS
 3: for node in reverse_nodes do
 4:   neighbors_node = CFG.out_neighbors(node)
 5:   for n_node in neighbors_node do
 6:     if n_node not in reverse_nodes then
 7:       check_node.append(node)
 8:     end if
 9:   end for
10: end for
11: $check\_line$ = get_line(check_node) // Find the line No. according to the node information
12:
13: return $check\_line$
```

Figure 6 is a sample program control flow graph of a potentially vulnerable file, introducing the process of applying Algorithm 1 to find its checkpoint. The Sink node in the figure is obtained by the *node_search* function. We used the BFS algorithm to find the set of all nodes in the CFG that can reach *target_node*, that is, recursively traverse the predecessor nodes. We found all the nodes that can reach Sink node are [$Sink$, BB_5, BB_4, BB_2, BB_1], which is called the set *reverse_nodes*.

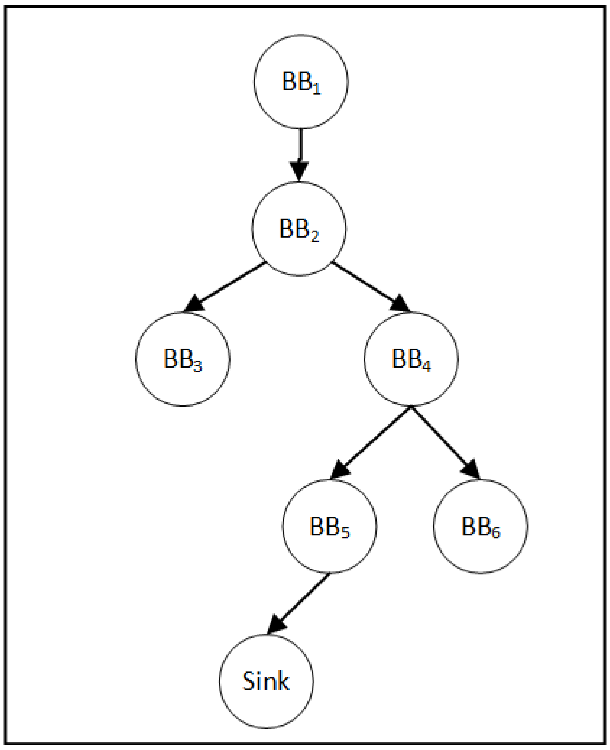


Figure 6. Sample control flow graph of potentially vulnerable files.

Finally we traversed all nodes in the set *reverse_nodes*. If there is a node and its successor node is not in the *reverse_nodes* set, then the node is a checkpoint. Among them, BB_2's successor BB_3 and BB_4's successor BB_6 are not in the *reverse_nodes* set, so the checkpoint set [BB_2, BB_4] is found.

Algorithm 2 Checkpoint instrumentation

```
Input: Check point located line No. : check_line, Path of potentially vulnerable file: path
1: file_content = file(path) // Read file by line
2: check_count = count(check_line)
3: for i = 0; i < check_count; i++ do
4:     line = check_line[i]-1;
5:     file_content[line] = "echo("Cefuzz check#".(i+1)."# checking...");\n".file_content[line]."echo("Cefuzz check#".(i+1)."# bypass!!!");\n";
6: end for
```

Algorithm 2 inserts characteristic strings before and after the statement where the checkpoint is located. When the checkpoint is encountered and successfully passed, the web application outputs *Cefuzz check #No.# checking* and *Cefuzz check #No.# bypass*; when the checkpoint cannot be passed smoothly, only *Cefuzz check #No.# checking* is output.

Listing 2 shows a snippet after sample code 1's checkpoint instrumentation. With such a checkpoint instrumentation, we can secure the current fuzzer's execution in that php file (by asking which checkpoint they are at or whether all checkpoints have been successfully broken). This fine-grained information is missing from the work of other researchers, and it is also important information to guide our seed mutation.

Listing 2. Slice of sample code 1 after checkpoint instrumentaion.

```
<?php
if( isset( $_POST[ 'Submit' ] ) ) {
// Get input
$target = $_REQUEST[ 'ip' ];
// check point1
echo("Cefuzz check\#1\# checking...");
if( strstr( $target, " " ) ){
exit("Find a space");
}
echo("Cefuzz check\#1\# bypass!!!");
// check point 2
echo("Cefuzz check\#2\# checking...");
if( strstr( $target, "&&" ) ){
exit("Find a &&");
}
echo("Cefuzz check\#2\# bypass!!!");
...
...
$html .= "<pre>{$cmd}</pre>";
}
?>
```

4.3. Seed Mutation

Seed mutation includes a **seed selection** module and a **seed mutation** module. The seed selection module evaluates the test cases after each round of fuzzing, and selects the seeds that perform better in each round to be mutated by the seed mutation module for the next round of fuzzing.

Seed selection and seed mutation are both key steps in our method. The performance of the seeds used in each round of fuzzing directly determines whether each round of fuzzing can be closer to the vulnerability located area and whether it is more likely to trigger the vulnerability. The performance of each test seed is determined based on the feedback provided in vulnerability validation. The relevant algorithms are described in Vulnerability Validation. In addition, seed mutations are also responsible for producing the initial seeds.

4.3.1. Initial Seed

The initial seed is the seed that has not mutated and is used for the first round of dynamic verification of vulnerabilities. The characteristic of the initial seed of the RCE vulnerability in this paper is that the command execution result can be displayed. Once the vulnerability is successfully triggered, it can be easily determined based on the HTTP response data packet or the change of the root directory of the web server.

Test cases are based on seed generation. Test cases for web application RCE vulnerabilities generally consist of multiple syntax blocks.

First, when the test case is inserted into a vulnerable position, the test case may need to close the original command execution statement to ensure that the test case has not a syntax error that causes the execution failure.

Secondly, after the test case closes the original command, it will try to execute its own attack payload. The attack payload is generally a complete command execution statement consisting of the executed command/function and related execution parameters.

Finally, the test case may need to use a closing character to mark the end of the entire command execution statement to ensure the integrity of the entire execution statement after the test case is inserted.

In summary, the test cases for RCE vulnerabilities can be expressed as:

$$Command\_case = [S], < C >, < P >, [E] \tag{1}$$

In this formula, S represents the closed syntax block of the original command execution statement in the test case (called **closed syntax unit**). As the command execution vulnerability does not necessarily require this syntax block, S is an optional syntax block.

C represents the name of the command or function executed by the attack payload in the test case (called **command name unit**). It is related to the execution result of the test case. It is a key part of the test case and belongs to the required syntax block.

P represents the execution parameter (called **command parameter unit**) used by the command or function used by the attack payload in the test case, which is related to the execution result of the test case and is also a required syntax block.

E represents the statement terminator of the command execution statement in the test case (called **statement end unit**). This grammar block is required under certain circumstances, so E is an optional grammar block.

Some initial seed samples are as follows:

(1) **system('echo Cefuzz > Cefuzz.php')**, in which the closed grammar unit and the sentence end unit are not used, the command parameter unit C is system, and the command parameter unit P is ('echo Cefuzz > Cefuzz.php')

(2) **print(md5(Cefuzz))**, in which the closed grammar unit and the end of sentence unit are not used, the command parameter unit C is print, and the command parameter unit P is (md5(Cefuzz))

(3) **phpinfo()**, in which the closed syntax unit and the sentence end unit are not used, the command parameter unit C is phpinfo, and the command parameter unit P is ()

(4) **curl http://xxx.xx.xx.xxx/Cefuzz.txt | bash**, in which the closed syntax unit and the end of sentence unit are not used, the command parameter unit C is curl, and the command parameter unit P is http://xxx .xx.xx.xxx/Cefuzz.txt | bash

The four sample seeds represent three different ways of display, respectively:

In sample 1 and sample 4, write a file named feature string (Cefuzz.php) in the web server directory, and the vulnerability trigger can be learned through **directory monitoring**.

Sample 2 outputs a string of specific character strings, the md5 value of Cefuzz (d9a6adadd9525a1bd3f8b274d6f22cdf), and the vulnerability trigger can be known by regular matching on HTTP response packets.

Sample 3 executes phpinfo(); phpinfo() will output the php environment variable information, and the vulnerability trigger can be known by regular matching on the HTTP response packet.

4.3.2. Seed Selection Rules

There are two types of seed selection rules, the first round of seed selection and the non-first round of seed selection.

The first round of seed selection is to randomly read the initial seeds. The non-first round of seed selection is based on the feedback information of the current round of vulnerability verification, in descending order of the number of checkpoints passed, and the seeds that pass the same number of checkpoints are sorted randomly.

Figure 7 shows the control flow graph of a sample code 1. The green dots represent checkpoints and the pink dots represent vulnerability trigger points. The more checkpoints passed, the test cases can reach the basic block closer to the sink point. Therefore, the more seeds that pass the checkpoint, the better the quality, and these seeds will be selected first for the next round of mutation.

4.3.3. Seed Mutation Operation

Vulnerability test seeds for RCE vulnerabilities require two conditions to be able to execute. One is that the command provided by the attacker conforms to the grammatical specification, and the other is that the command provided by the attacker can still take effect after the possible sterilization of the web application.

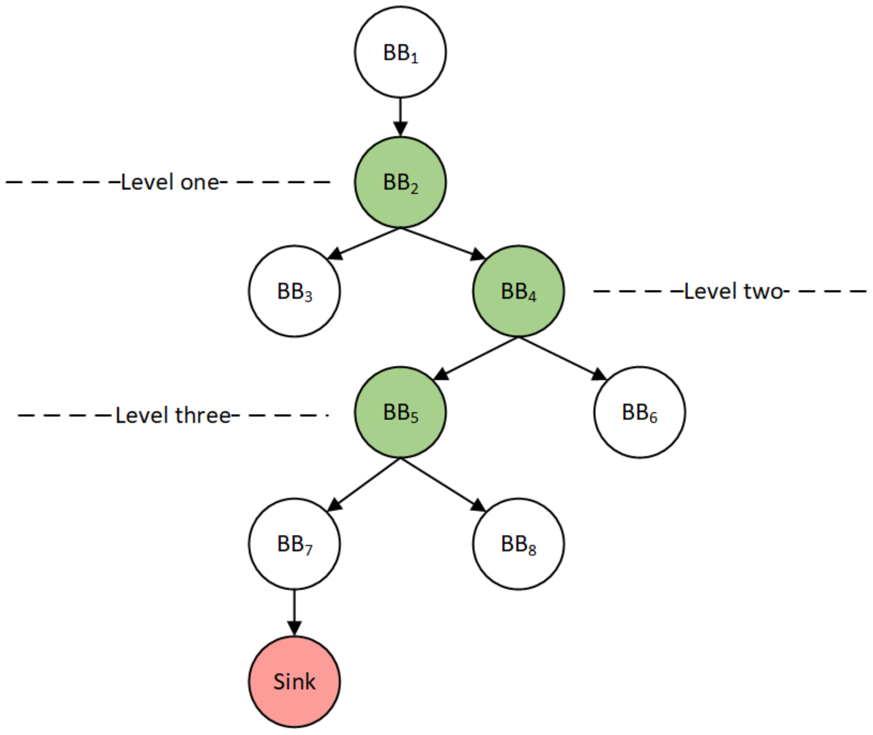


Figure 7. CFG of sample code 1.

The current defense against RCE vulnerabilities is blacklist defense, matching and identifying some dangerous functions. Therefore, the seed mutation operation considers the mutation to meet the grammatical specification or bypasses the blacklist defense so that the test seed can be executed.

Consequently, there are six kinds of seed mutation operations designed in our method, which are **adding prefix(M1)**, **adding suffix(M2)**, **replacing space(M3)**, **keyword bypassing(M4)**, **case bypassing(M5)**, and **encoding bypassing(M6)**.

RCE vulnerabilities generally require splicing the command payload provided by the attacker to the command/code execution function in the source code, so the M1 and M2 mutation operations can increase the probability of the mutated seed meeting the grammatical specification, which is aimed at the **closed syntax unit** and the mutation operation of **statement end unit**. M3, M4, and M5 are mutation operations for the **command name unit**. M3 and M4 operations replace spaces and insert separators that do not affect the execution of the function. M5 operation rewrites the case of commonly known dangerous function names, increasing the probability of bypassing the sterilization. The M6 operation encodes the content of the seed to increase the probability of bypassing the sterilization for feature functions and feature strings. It is a mutation operation for the **command parameter unit**.

The M1 operation adds a prefix to the seed string. The added string includes backquote (`), single quotation mark (′), double quotation mark ("), semicolon (;), single quotation mark comma (′,), double quotation mark comma (",) , pipe command (|), right square bracket semicolon (];), left angle bracket question mark (<?), and command connector (&&) etc.

The M2 operation adds a suffix to the seed string. The added string includes backquote (`), single quotation mark (′), double quotation mark (") , semicolon (;), single quotation mark comma (′,), double quotation mark comma (",) , semicolon double backslash (;//), question mark right angle bracket (?>), semicolon question mark right angle bracket (;?>), semicolon question mark right angle bracket, and double backslash (;? >//), etc.

The M3 operation is to replace the spaces in the seed string with special strings. Some special strings need to be used with special prefixes and suffixes. Special strings have left angle brackets (<), \$IFS, %PROGRAMFILES: 10, −5%, etc. Special suffixes include left and right curly braces (), a=\$',’&&a, etc.

The M4 operation is to extract the system command string from the seed string, such as whoami, echo, cat, etc., and insert special characters in it. Special characters include single quote ('), double quote ("), backslash (\\), dot (.), and \$@, etc. Inserting these special characters can bypass the characteristic command string matching and does not affect the normal execution of the command.

The M5 operation is to extract the dangerous function names in the seed string, such as system, eval, phpinfo, etc., and rewrite them with random character case, such as rewriting system to SysTem, bypassing case-sensitive feature function matching, and does not affect the normal execution of dangerous functions.

The M6 operation is to encode the seed content. For example, the seed content is $eval(\$\_GET[a])$, and the $\$\_GET[a]$ is encoded. The encoding algorithm used needs to have a corresponding decoding algorithm. If the string is encoded in hexadecimal form, the specific operation is to encode $eval(\$\_GET[a]);$ in hexadecimal in advance to obtain the string $6576616c28245f4745545b615d293b$. The encoded string is put into the decoding algorithm, the seed content obtained is $\$hex = hex2bin(``6576616c28245f4745545b615d293b");$ $eval(\$hex);$ the final code executed is actually $eval(`eval(\$\ _GET[a]);')$, which can bypass the detection of $\$\_GET$.

4.3.4. Seed Mutation Strategy

Seed mutation strategy refers to how to select a seed mutation operation to mutate a given seed. As each seed mutation operation is designed to bypass a certain content filtering check, the combination of multiple mutation strategies can increase the probability of bypassing multiple checkpoints. The seed mutation strategy selected in our method is the sum of the binomial coefficients of all the six mutation strategies of M1, M2, M3, M4, M5, and M6. There are 64 mutation strategies including the empty set.

In the first round of mutation, the empty set mutation is performed first, that is, no mutation operation is performed. The second round of mutation uses M1, M2, M3, M4, M5, and M6 to mutate separately, and record the test case passing the checkpoint after the mutation. For example, if the M1 mutation passed the most checkpoints, then in the next round of mutation, on the premise that M1 has been used, we add other mutations (M2–M6) operations. The schematic diagram of mutation strategy selection is shown in Figure 8, where T_1–T_4 represent the first to fourth rounds of testing, respectively. M1 performs better, meaning that the seed of M1 mutation can pass more checkpoints and successfully verify the vulnerabilities after four rounds of testing.

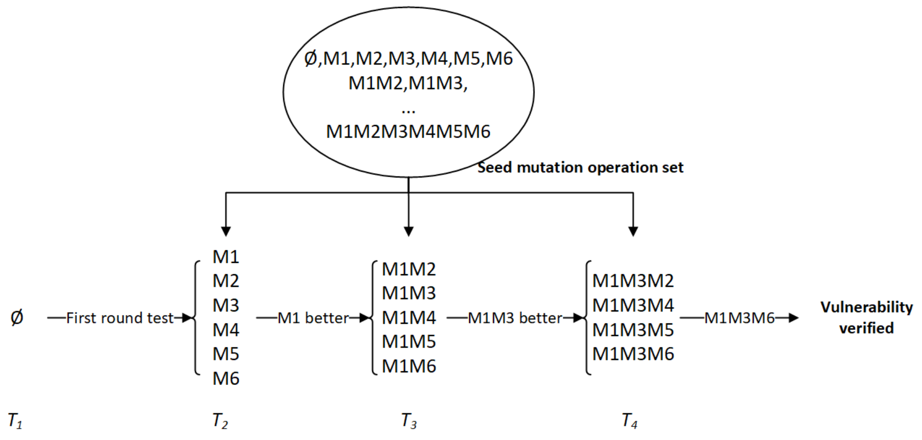


Figure 8. An example of seed mutation strategy.

4.4. Vulnerability Verification

Vulnerability verification includes two parts: **vulnerability verification** and **information feedback**. Vulnerability verification relies on special commands for initial seed, and information feedback relies on checkpoint instrumentation.

As mentioned in Section 4.3, there are three types of seeds in the initial seed: writing feature files, output feature strings, and output phpinfo. Correspondingly, vulnerability verification uses three methods for verification: **signature file monitoring**, **characteristic string matching**, and **phpinfo output matching**. Algorithm 3 is the vulnerability verification algorithm.

Algorithm 3 Vulnerability verification

```
Input: Response packet content: content, Webapp source code path: path
Output: Bool true/false, indicates that the vulnerability exists or does not exist yet
1: file_hash_array_old= file_hash_array(path)
2: ... // Send test request
3: file_hash_array_new= file_hash_array(path)
4: diff_array = array_diff_fast(file_hash_array_new,file_hash_array_old)
5: phpinfo_pattern = "/Zend Memory Manager/"
6: pattern = "/d9a6adadd9525a1bd3f8b274d6f22cdf/"
7: if (file_monitor($diff_array) or vul_match(content,phpinfo_pattern) or
   vul_match(content,pattern)) then
8:
9:     return TRUE
10: else
11:
12:     return FALSE
13: end if
```

Characteristic file monitoring is mainly completed by file_hash_array function and file_ monitor function. Before the test starts, the file_hash_array function traverses all PHP files in the path and calculates the MD5 hash of the file which save into file_hash_array_old array. We repeated the operation after each test request is sent and saved the result in file_hash_array_new array. The File_monitor function compares the two hash arrays to determine whether any change happened and then matches the changed filename. The existence of a new filename that matches the characteristic string indicates that the vulnerability verification is successful.

Characteristic string matching, phpinfo output matching performs pattern matching on response packets. The characteristic string refers to a fixed md5 value output ($d9a6adadd9525a1bd3f8b274d6f22cdf$), the characteristic string output by phpinfo ($ZendMemoryManager$), matching these two characteristic characters indicates that the vulnerability verification is successful.

Similar to the detection of characteristic strings in **vulnerability verification**, **information feedback** also uses pattern matching to identify the number of checkpoints passed, as shown in Algorithm 4, and attaches a mark of the number of checkpoints to the seed and feeds it back to **seed mutation**.

Algorithm 4 Checkpoint pass detection

```
Input: Response packet content: content, Webapp source code path: path, Total number of
    checkpoints: count
Output: Number of checkpoints passed: bypass_count, If the number is equal to 999, all
    the checkpoints have passed
 1: checking_pattern = "/Cefuzz check#(.*)# checking/"
 2: bypass_pattern = "/Cefuzz check#(.*)# bypass/"
 3: checking_count = match_all_count(checking_pattern,content) // Checkpoint encoun-
    tered
 4: bypass_count = match_all_count(bypass_pattern,content) // Checkpoints that have
    passed
 5: if (if(bypass_count < count)) then
 6:
 7:     return bypass_count
 8: else
 9:
10:     return 999
11: end if
```

4.5. Vulnerability Verification Code Output

When a vulnerability is discovered, the vulnerability verification code extracts the successfully verified packet request parameters and requests data. It then uses them to replace some parameters on PoC template file. Listing 3 shows the vulnerability verifying code template file, where ##*get_data*##, ##*url*##, ##*cookie_data*##, and ##*post_data*## are the anchor points for replacement of GET request data, target URL, cookie data, and POST request data, respectively.

Listing 3. PoC template file.

```php
<?php
$get_param = ##get_data##;
$url = ##url##."?".$get_param;
$cookie_data = ##cookie_data##;
$content = urldecode(http_build_query(##post_data##));
$content_length = strlen($content);
$options = array(
'http' => array(
'method' => 'POST',
'header' =>
"Content-type: application/x-www-form-urlencoded\r\n".
"Accept-Encoding: deflate\r\n".
"Cookie: $cookie_data".
"Content-length: $content_length",
'content' => $content,
)
);
$request = file_get_contents($url, false,
stream_context_create($options));
?>
```

5. Evaluation

In order to evaluate the automated verification of web applications based on directed fuzzing proposed in this paper, we designed a series of experiments on real programs and compared them with related technologies. The experiments are described below.

5.1. Evaluation Setup

We designed the experiments to answer the following research questions:

RQ1: Using our method to test the simple vulnerability examples in the vulnerability practice platform, can it automatically generate PoC?
RQ2: Can our method achieve a good verification effect on known vulnerabilities in real programs?
RQ3: Can our method detect unknown vulnerabilities in real programs?

The first experiment is used to verify whether our method has the basic RCE vulnerability detection ability and the PoC code generation ability, the second experiment is used to verify the verification ability of our method against known vulnerabilities, and the third experiment is used to verify our method's ability to detect unknown vulnerabilities.

Experimental environment: All experiments are run on a machine equipped with Intel Core i7-10875H processor, which contains four logical cores of 2.30 GHz, 16 GB memory, and the operating system is 64-bit Windows 10 20H2 or 64-bit Linux Ubuntu 18.04, mainly used The PHP version is 7.1.24, adjusted according to the PHP version required by the target web applications.

5.2. Evaluation Benchmarks

The datasets of the three experiments are shown in Table 1. Experiment 1 uses the three vulnerability practice platforms DVWA, bWAPP, and Pikachu to verify the preset RCE vulnerabilities among them. Experiment 2 uses seven RCE vulnerabilities publicly available on the Internet, including Pbootcms, cmsuno, MyuCMS, FineCms, Maccms, and Seacms. In Experiment 3, 10 open source PHP CMS were selected, including CREMB, imcat, WMCMS, and xiaohuanxiongcms published by Chinese authors. and Maxsite cms, GetSimple, kodicms, ph7cms, symphonycms, and GreenCMS published by authors from other countries.

Table 1. Experimental data sets.

Type of Experiment	Target Web Application	Version	Vul-ID
RQ1	DVWA [22]	1.9	–
	bWAPP [23]	2.2	–
	Pikachu [24]	1.0	–
RQ2	Zzzcms [25]	2.03	CVE-2021-32605
	Pbootcms [26]	1.3.2	CNVD-2021-33224
	cmsuno [27]	1.6.2	CVE-2020-25557
	MyuCMS [28]	2.1	CNVD-2019-43630
	FineCms [29]	5.0.9	CVE-2017-11585
	Maccms [30]	8.x	CVE-2017-17733
	Seacms [31]	6.28	–
RQ3	CREMB [32]	2.6.13	–
	CREMB [32]	3.1.0	–
	imcat [33]	5.2	–
	WMCMS [34]	4.346	–
	xiaohuanxiongcms [35]	5.0	–
	Maxsite cms [36]	108	–
	GetSimple [37]	3.3.16	–
	Kodicms [38]	13.82.135	–
	ph7cms [39]	16.2.2	–
	symphonycms [40]	2.7.10	c–
	GreenCMS [41]	2.3.0603	–

Furthermore, we have selected two currently popular web application static detection systems, RIPS [42] and Seay [43], as experimental comparison tools. These two tools cannot automatically verify vulnerabilities. Therefore, during the experiment, as long as the tool detects the corresponding vulnerabilities in the source code files, we think that the tool successfully verified the vulnerability. Web application fuzzing tools, such as ffuf [44], wfuzz [45], vaf [46], etc., are essentially just API brute force traversal tools and do not have an important feedback adjustment mechanism in the fuzzing, so they are not used for comparison during the experiment.

5.3. Effectiveness Test of Vulnerability Verification and PoC Generation (RQ1)

In order to evaluate the basic ability of our method for vulnerability verification and PoC generation, we selected RCE vulnerabilities on three vulnerability training platforms for experiments. The experimental results are shown in Table 2.

Table 2. Basic ability evaluation experiment results.

Target Web Applications	**RIPS**	**Seay**	**Cefuzz**
DVWA	✓	✓	✓
bWAPP	✓	✓	✓
Pikachu	✓	✓	✓

The experimental results show that RIPS, Seay, and Cefuzz can all detect the preset RCE vulnerabilities in all vulnerability practice platforms, and Cefuzz can generate usable PoC.

5.4. Verification Ability Experiment for Enown Vulnerabilities (RQ2)

In order to evaluate the verification ability of our method for known vulnerabilities, the second experiment selected seven known vulnerabilities for experiments. The results of Experiment 3 are shown in Table 3. RIPS detected two known vulnerabilities, Seay detected four known vulnerabilities, and Cefuzz was able to detect all seven vulnerabilities.

Table 3. Known vulnerabilities verification experiment results.

Target Web Applications	RIPS	**Seay**	**Cefuzz**
zzzcms	✗	✗	✓
Pbootcms	✗	✓	✓
cmsuno	✓	✓	✓
MyuCMS	✗	✓	✓
FineCms	✗	✗	✓
Maccms	✗	✓	✓
Seacms	✓	✗	✓

RIPS uses context-sensitive taint analysis technology for vulnerability detection. Seay's technology principle is to perform keyword regular matching on each source code file of the target web application, while Cefuzz conducts directed fuzzing based on the vulnerable code-related information obtained by static analysis.

Therefore, as long as Seay recognizes its defined malicious function or malicious regular pattern, it will report it as a vulnerability. The core of RIPS is taint analysis technology. Once RIPS' static analysis finds that the vulnerability may not be reachable, it will not report the vulnerability.

For example, the cause of the vulnerability in Maccms is an eval function splicing user input data. Seay simply scans the eval function and variables here, but RIPS is not regarded as a vulnerability because it cannot analyze that user input spreads to the eval function, which leads to more vulnerabilities detected by Seay than RIPS in this experiment.

Compared with RIPS, Cefuzz's static vulnerability analysis module also uses taint analysis technology, but due to the addition of inter-process analysis and taint analysis during the implementation process, the static analysis results of Cefuzz are more precise than RIPS, ensuring that the vulnerable code information contains vulnerability information. In the directed fuzzing stage, due to the instrumentation and improved seed mutation mechanism, the efficiency of fuzzing is improved and, finally, Cefuzz can detect more vulnerabilities than RIPS.

5.5. Evaluation Experiment of Detection Ability for Unknown Vulnerabilities (RQ3)

In order to evaluate our method's ability to detect and discover unknown vulnerabilities, we selected 10 popular web applications for testing, and finally found 13 unknown RCE vulnerabilities. The details of these vulnerabilities have been communicated to the authors of the corresponding codes and submitted to the China National Vulnerability Database (CNVD). Table 4 gives information on the unknown vulnerabilities discovered, including version, vulnerability type, and vulnerability number of the web application to which the vulnerability belongs.

Table 4. Unknown vulnerability discovered by Cefuzz.

Web Applications	Version	Types	Vul-ID
CREMB	2.6.13	Command Injection	CNVD-2020-31496
CREMB	3.1.0	Command Injection	CNVD-2020-31078
imcat	5.2	Command Injection	CNVD-2020-33075
WMCMS	4.346	Code Execution	CNVD-2020-31070
xiaohuanxiongcms	5.0	Code Execution	CNVD-2021-80217
xiaohuanxiongcms	5.0	Code Execution	CNVD-2021-80221
xiaohuanxiongcms	5.0	Code Execution	CNVD-2021-80223
Maxsite cms	108	Code Execution	CNVD-2021-80212
GetSimple	3.3.16	Code Execution	CNVD-2020-62880
KodiCMS	13.82.135	Code Execution	CNVD-2021-79629
ph7cms	16.2.2	Code Execution	CNVD-2021-79627
symphonycms	2.7.10	Code Execution	CNVD-2021-81498
GreenCMS	2.3.0603	Code Execution	CNVD-2021-81522

Let us take CREMB as an example to introduce the process by which Cefuzz found one of the unknown vulnerabilities, CNVD-2020-31078. CREMB is a web application used to quickly build an online shopping system. It was once rated as the most valuable open source project of Gitee in 2019 by Gitee (Internet code hosting website). Cefuzz was used to detect it and found two vulnerabilities.

The vulnerability CNVD-2020-31078 is a command injection vulnerability hidden in CREMB. Attackers can trigger the vulnerability and obtain server management permissions by sending constructed data packets. Listing 4 shows the code snippet of the vulnerability. As the details of the vulnerability have not been made public, part of the information is hidden in the code snippet below.

As shown in Listing 4, the trigger point of the vulnerability is in the *bbb* function in b.php. Cefuzz first identifies the function parameter as the source of taint. In line 47, the formal parameter $text is passed into the dangerous function fwrite without any cleaning function processing, so the custom function is judged as a dangerous function by Cefuzz and stored in the mysql database. When the web application calls the *aaa* function in a.php, the function calls the post function to receive user input on lines 68 and 69, and is judged as the source of taint, with $a and $b as tainted variables. In line 75, these two taint variables are passed into the dangerous function *bbb* without any cleaning function, that is, the taint passes into the taint convergence point, and has since been positioned as a vulnerable code by the static taint analysis module.

Listing 4. CNVD-2020-31078 Snippet.

```
67     public function aaa(){
68             $a = post('parameter_a');
69             $b = post('parameter_b');
70             if(!empty($comment) && !empty($filepath)){
71                     ...
72                     ...
73                     if(strstr($uname,'Windows')!==false)
74                     ...
75                     $res = class::bbb($b,$a);
76                     if($res){
77                             ...
78                     }else{
79                             ...
80                     }
81             }
...
85       }
(a) a.php

42       static function bbb($filename, $text, $openmod='w')
43       {
44             if(@$fp = fopen($filename, $openmod))
45             {
...
47                     fwrite($fp, $text);
...
50             }
...
55       }
(b) b.php
```

After the vulnerability automatic verification module receives the information about the vulnerable code, it first performs checkpoint instrumentation on the web application source code, namely line 70 of a.php and line 44 of b.php. Then, request parameters on lines 68 and 69 are extracted, as well as the parameters parameter_a and parameter_b that need to be submitted in post. The two parameters respectively receive the seed information provided by the **seed mutation** and perform fuzzing. When the Cefuzz.php file is detected in the web application directory, it proves that the vulnerability exists.

5.6. Discussion

In this section, we will discuss the limitations of Cefuzz and future directions of research to further improve the completeness of Cefuzz.

Although our fuzzing method improves the feedback information mechanism in fuzzing technology and improves the effectiveness, it is still impossible to verify all vulnerabilities. The main limitations of our method are as follows:

(1) The result of static analysis determines the scope of our fuzzing method. However, many vulnerabilities need to be combined with dynamic characteristics to be judged, so our method is not complete enough. In future research, we will consider how to add a monitoring mechanism for the runtime state of the program and incorporate the runtime state into the feedback information.

(2) While our method proposes six mutation operations, due to the different habits of web application developers, these six mutation operations cannot meet the command splicing requirements in various situations. Therefore, in future research, we will consider how to automatically obtain the syntax of command splicing in different vulnerability scenarios and adaptively mutate it.

(3) Our method mainly solves the problems in the verification of RCE vulnerabilities, but for other vulnerabilities we only need to expand our initial seed and add suitable mutation operations to solve the verification problems. We will try to expand the types of vulnerabilities we can analyze in future research.

(4) Our method is to use the directed fuzzing method to verify the known vulnerabilities in the web application based on the source code and find the unknown vulnerabilities in the web application. Our research object is the security of the web application itself. The mentioned checkpoints are also a defense method for the web application source code itself. Therefore, the PoC generated by our method may fail on Internet web applications as many Internet servers will set up web application firewalls or enable some PHP security modules. These security modules are not used in the experimental part of this paper, because this is beyond the scope of this paper. Of course, our method can generate a PoC that bypasses these protection methods. It only needs to regard the security component as a checkpoint and feedback the alert information by the security component to the seed mutation module. The mutation mechanism can generate a payload that bypasses the security component after multiple rounds of mutation.

Both ModSecurity [47] and Suhosin [48] are security components that perform blacklist defense based on rule matching. The advantage of the blacklist is its ease of use and scalability. Administrators only need to update the rule base in time to block known attack payloads. However, on the Internet, there may be new attack payloads that can bypass the rule base every day, and administrators cannot have a comprehensive and timely updated rule, which leads the blacklist's ability to withstand new attack payloads to be almost zero. Using WAF or PHP security plug-ins can indeed alleviate attacks against web applications to a certain extent, but this does not mean that there are no vulnerabilities in web applications. On the contrary, vulnerabilities in web applications are covered up and the security of web applications is not improved. Cefuzz can bypass blacklist defense. The current mainstream blacklist feature words can be bypassed through the six mutation strategies or combinations of strategies mentioned in this paper. In future research, we may carry out research on automated understanding and bypassing of WAF defense rules.

6. Related Works

In this section, some static and dynamic detection methods are introduced.

6.1. Static Methods

Livshits et al. [11,49] proposed a static analysis method based on scalable and precise point-to-point analysis. A user-defined vulnerability specification is provided by the user. This static analysis method can find all vulnerabilities that meet the specification, and the detection is very accurate. However, if the specifications provided by the user are incomplete or there are errors, it will lead to false negatives and false positives. The detection effect depends on the user's experience and the degree of automation is low.

In order to improve the automation of web application detection, Jovanovic et al. [50] developed a web application detection tool Pixy based on taint analysis technology. This tool has contributed a lot of work to PHP alias modeling, but the number of web applications using PHP aliases is limited. Only 29 built-in dangerous functions are configured, resulting in a high false-positive rate for this tool.

Since then, Dahse et al. [42] has proposed a new static code analysis tool called RIPS. This tool constructs an abstract syntax tree (AST) for each PHP file in a web application, converts the AST to a control flow graph (CFG), and conducts vulnerability detection on web applications through context-sensitive taint analysis. Unlike Pixy, this tool further enriches the dangerous function library and uses AST for context-sensitive analysis, which reduces the false positive rate. However, due to the inherent shortcomings of static analysis technology, the runtime state is ignored, so it still has a high false-positive rate.

Compared with our work, the above three works, first of all, are relying on expert experience, have a high false-negative rate, a lack of a dynamic verification environment, resulting in the output of the judgment results needing to be manually verified again, and finally suffer from poor practicality.

6.2. Dynamic Methods

Dynamic analysis can also be combined with dynamic taint tracking, using a set of specific inputs to observe the running status of web applications [51].

O.van Rooij et al. [52] designed and implemented a coverage-based web application gray box fuzz testing tool, in which they staked code in each basic block of the source code so that, during the fuzzing, the basic block where the current execution was located could be obtained. They also created a code analysis tool for JavaScript to analyze whether an alert function call was generated during each fuzzing, and judged whether the XSS vulnerability was successfully triggered based on this condition. Compared with our work, their work is not guided, and the fuzz tester is blindly tested. In addition, their payload mutation mechanism is too limited, essentially using a brute force traversal using an attack payload that is effective in the wild. Another point is that their vulnerability trigger judgment mechanism is expensive and requires additional code analysis, and our work is to design the payload; if the payload is executed successfully, it can be easily monitored by our fuzzer.

Huang et al. [53] proposed a fuzzing system for detecting PHP web application file upload vulnerabilities, which generates executable code templates that summarize the vulnerability-related semantics of the server-side web application through static analysis, and then places these templates in the PHP running environment, and uses fuzzing to detect vulnerabilities in executable code templates. This work is consistent with our work, which is essentially a gray box fuzzing, but he uses static analysis to extract semantic templates for testing. There is a problem of semantic loss, one is that the semantic template that may be extracted makes the original vulnerability disappear, and the other is that the semantic template that may be extracted has caused a new vulnerability, which makes us unable to be sure that the vulnerability detected on the template is a real vulnerability, and it needs to be manually verified once. Our work no longer requires manual verification after fuzzing.

Taekjin et al. [54] proposed a penetration testing system FUSE for identifying file upload vulnerabilities. The system uses dynamic analysis technology to send test requests when the web application is running, and check whether the file upload is successful through the file monitoring module. FUSE conducted a detailed analysis of bypassing file upload vulnerability defenses and built a semi-automated dynamic analysis framework. Its disadvantage is that the degree of automation is too low, and the user needs to provide information including the vulnerability URL, the parameter name to be sent, the account password, etc. FUSE is less automated than we work, a lot of information needs to be provided through configuration files, and the usefulness is average.

Benediktd et al. [55] implemented VeriWeb, a tool for automatically discovering and systematically exploring the execution path of a website that users can follow in a web application. VeriWeb can automatically browse the dynamic components of the website, including form submission and client-side script execution. Whenever a new web page is checked, the system will determine that all possible actions of the user can be performed through buttons with JavaScript handlers or through form submissions and that they can be performed in a systematic way. When encountering a form, VeriWeb uses SmartProfiles to identify the values that should be entered into the form.

Huang et al. [56] proposed a black-box fuzzing method to automatically test SQL injection vulnerabilities in web applications in real scenarios. The integrated and simulated the functions of the web browser in the crawler, enabling it to test web applications in real scenarios. The crawler uses a "full crawl" mechanism to reverse engineer the web application to identify all data entry points. Then, with the help of self-learning to inject the knowledge base, fault injection technology is used to detect SQL injection vulnerabilities. The knowledge base selects the best injection mode based on the experience learned from the previous injection feedback, and then expands the knowledge base as more pages are crawled. Both previous experience and knowledge expansion can help produce better injection patterns. They also proposed a novel return data packet analysis algorithm to help crawlers interpret the content of the injected return data packet.

FLAX [57] is a taint-enhanced black-box fuzzing technology that targets client-side input validation vulnerabilities in JavaScript code and DOM-based XSS attacks. In particular, it applies dynamic taint analysis to extract knowledge about receivers in JavaScript code and then uses it to trim the input variation space and guide effective fuzzing.

7. Conclusions

RCE vulnerabilities are an important threat to cyberspace security. Attackers can obtain the highest authority of the target system through the RCE vulnerability which may lead to tremendous damage. To this end, discovering RCE vulnerabilities and increasing the automation of RCE vulnerabilities detection are the main challenges of current RCE vulnerability detection. This paper proposes a PHP RCE vulnerability directed fuzzing method, combining the static analysis technology and dynamic verification technology. This method is guided by the result of static taint analysis and performs instrumentation for the source code of the web application to increase the amount of feedback information. Then, based on the number of checkpoints bypassed by the test seeds, this method chooses an optimal mutation strategy to improve the effectiveness of test seeds, and finally output the vulnerability PoC.

According to this method, we implemented a PHP RCE vulnerability directed fuzzing prototype system called Cefuzz. Experimental results show that Cefuzz's verification effect on known RCE vulnerabilities is better than the current web application vulnerability discovering technology, and 13 unknown vulnerabilities are found in 10 popular web CMSes.

Author Contributions: Conceptualization, Y.L. and J.Z.; methodology, J.Z., K.Z. and Z.C.; software, J.Z. and H.H.; validation, J.Z. and K.Z.; investigation, J.Z.; resources, Y.L.; data curation, J.Z. and K.Z.; writing—original draft preparation, J.Z.; writing—review and editing, Y.L., Z.C., H.H. and J.Z.; supervision, Y.L.; project administration, Y.L. All authors read and agreed to the published version of the manuscript.

Funding: This research was supported by the National Key Research and Development Project of China (No. 2017YFB0802900).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Acknowledgments: We would like to sincerely thank the reviewers for their insightful comments, which helped us improve this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Netcraft. August 2021 Web Server Survey. 2021 [EB/OL]. Available online: https://news.netcraft.com/archives/2021/08/25/august-2021-web-server-survey.html (accessed on 1 September 2021).
2. Roy, J.; Ramanujan, A. Understanding web services. *IT Prof.* **2001**, *3*, 69–73. [CrossRef]
3. CWE. Command Injection. [EB/OL]. Available online: https://cwe.mitre.org/data/definitions/78.html (accessed on 1 September 2021).
4. Wikipedia. Code Execution. [EB/OL]. Available online: https://en.wikipedia.org/wiki/Arbitrary_code_execution (accessed on 1 September 2021).

5. w3techs. Usage Statistics of Server-Side Programming Languages for Websites. [EB/OL]. Available online: https://nvd.nist.gov/vuln/search/statistics (accessed on 1 September 2021).
6. Chess, B.; McGraw, G. Static analysis for security. *IEEE Secur. Priv.* **2004**, *2*, 76–79. [CrossRef]
7. Ayewah, N.; Pugh, W.; Hovemeyer, D.; Morgenthaler, J.D.; Penix, J. Using Static Analysis to Find Bugs. *IEEE Softw.* **2008**, *25*, 22–29. [CrossRef]
8. Kals, S.; Kirda, E.; Kruegel, C.; Jovanovic, N. Secubat: A web vulnerability scanner. In Proceedings of the 15th International Conference on World Wide Web, Edinburgh, UK, 23–26 May 2006; pp. 247–256.
9. Fan, J.; Gao, P.; Shi, C.C.; Li, N.G. Research on combine White-box testing and Black-box testing of Web Applications security. In *Advanced Materials Research*; Trans Tech Publications Ltd.: Freinbach, Switzerland, 2014; Volume 989, pp. 4542–4546.
10. Denning, D.E. A lattice model of secure information flow. *Commun. ACM* **1976**, *19*, 236–243. [CrossRef]
11. Wassermann, G.; Su, Z. Static detection of cross-site scripting vulnerabilities. In Proceedings of the 2008 ACM/IEEE 30th International Conference on Software Engineering, Leipzig, Germany, 10–18 May 2008; pp. 171–180. ISSN 1558-1225. [CrossRef]
12. Miller, B.P.; Fredriksen, L.; So, B. An Empirical Study of the Reliability of UNIX Utilities. *Commun. ACM* **1990**, *33*, 32–44. [CrossRef]
13. Canakci, S.; Delshadtehrani, L.; Eris, F.; Taylor, M.B.; Egele, M.; Joshi, A. DirectFuzz: Automated Test Generation for RTL Designs using Directed Graybox Fuzzing. In Proceedings of the 2021 58th ACM/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 5–9 December 2021; pp. 529–534. [CrossRef]
14. Alhuzali, A.; Gjomemo, R.; Eshete, B.; Venkatakrishnan, V. {NAVEX}: Precise and scalable exploit generation for dynamic web applications. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 377–392.
15. Zhang, G.; Zhou, X.; Luo, Y.; Wu, X.; Min, E. PTfuzz: Guided Fuzzing With Processor Trace Feedback. *IEEE Access* **2018**, *6*, 37302–37313. [CrossRef]
16. Godefroid, P. Random testing for security: blackbox vs. whitebox fuzzing. In Proceedings of the 2nd International Workshop on Random Testing: Co-Located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), Atlanta, Georgia, 6 November 2007; p. 1.
17. Rawat, S.; Mounier, L. Offset-aware mutation based fuzzing for buffer overflow vulnerabilities: Few preliminary results. In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, IEEE, Berlin, Germany, 21–25 March 2011; pp. 531–533.
18. Miller, C.; Peterson, Z.N. *Analysis of Mutation and Generation-Based Fuzzing*; Independent Security Evaluators, Tech. Rep; Independent Security Evaluators: Baltimore, MD, USA, 2007; Volume 4.
19. Fielding, R.; Gettys, J.; Mogul, J.; Frystyk, H.; Masinter, L.; Leach, P.; Berners-Lee, T. Hypertext Transfer Protocol–HTTP/1.1. 1999. Available online: https://www.hjp.at/doc/rfc/rfc2616.html (accessed on 1 September 2021).
20. Chen, P.; Liu, J.; Chen, H. Matryoshka: fuzzing deeply nested branches. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, London, UK, 11–15 November 2019; pp. 499–513.
21. Zhao, J.; Lu, Y.; Wang, X.; Zhu, K.; Yu, L. WTA: A Static Taint Analysis Framework for PHP Webshell. *Appl. Sci.* **2021**, *11*, 7763. [CrossRef]
22. Digininja. DVWA. Available online: https://github.com/digininja/DVWA (accessed on 1 September 2021).
23. bWAPP. Available online: http://www.itsecgames.com/index.htm (accessed on 1 September 2021).
24. Zhuifengshaonianhanlu/Pikachu. Available online: https://github.com/zhuifengshaonianhanlu/pikachu (accessed on 1 September 2021).
25. Zzzcms. Available online: http://www.zzzcms.com/index.html (accessed on 1 September 2021).
26. Pbootcms. Available online: https://www.pbootcms.com/ (accessed on 1 September 2021).
27. boiteasite/cmsuno. Available online: https://github.com/boiteasite/cmsuno (accessed on 1 September 2021).
28. MyuCMS. Available online: http://www.myucms.com/ (accessed on 1 September 2021).
29. FineCms. Available online: http://xnxl.down.easck.com:880/code/2017/0724/finecms_v5.0.10.zip (accessed on 1 September 2021).
30. Maccms. Available online: https://www.maccms.cn/down.html (accessed on 1 September 2021).
31. Seacms. Available online: http://xnxl.down.easck.com:880/code/2016/0822/seacms_v6.28.zip (accessed on 1 September 2021).
32. ZhongBangKeJi. CRMEB. Available online: https://gitee.com/ZhongBangKeJi/CRMEB (accessed on 1 September 2021).
33. Peacexie/imcat. Available online: https://github.com/peacexie/imcat (accessed on 1 September 2021).
34. WMCMS. Available online: http://www.weimengcms.com/index.html (accessed on 1 September 2021).
35. Hiliqi/xiaohuanxiongcms. Available online: https://github.com/hiliqi/xiaohuanxiong (accessed on 1 September 2021).
36. Maxsite Cms. Available online: https://github.com/maxsite/cms (accessed on 1 September 2021).
37. GetSimple. Available online: http://get-simple.info/ (accessed on 1 September 2021).
38. Kohana. KodiCMS. Available online: https://github.com/KodiCMS-Kohana/cms (accessed on 1 September 2021).
39. pH7Software. ph7cms. Available online: https://github.com/pH7Software/pH7-Social-Dating-CMS (accessed on 1 September 2021).
40. Symphonycms. Available online: https://github.com/symphonycms/symphonycms (accessed on 1 September 2021).
41. GreenCMS. Available online: https://github.com/GreenCMS/GreenCMS/ (accessed on 1 September 2021).

42. Dahse, J.; Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. *NDSS* **2014**, *14*, 23–26.
43. Seay. Available online: https://github.com/f1tz/cnseay (accessed on 1 September 2021).
44. Tomikoski. Fast Web Fuzzer. Available online: https://github.com/ffuf/ffuf (accessed on 1 September 2021).
45. Xmendez. Wfuzz–The Web Fuzzer. Available online: https://github.com/xmendez/wfuzz (accessed on 1 September 2021).
46. d4rckh. Very Advanced Fuzzer. Available online: https://github.com/d4rckh/vaf (accessed on 1 September 2021).
47. SpiderLabs. ModSecurity. Available online: https://github.com/SpiderLabs/ModSecurity (accessed on 20 December 2021).
48. Xmendez. Suhosin. Available online: https://github.com/sektioneins/suhosin (accessed on 20 December 2021).
49. Livshits, V.B.; Lam, M.S. Finding Security Vulnerabilities in Java Applications with Static Analysis. *USENIX Secur. Symp.* **2005**, *14*, 18.
50. Jovanovic, N.; Kruegel, C.; Kirda, E. Pixy: A static analysis tool for detecting Web application vulnerabilities. In Proceedings of the 2006 IEEE Symposium on Security and Privacy (S P'06), Oakland, CA, USA, 21–24 May 2006; ISSN 2375-1207. [CrossRef]
51. d'Amore, F.; Gentile, M. *Automatic and Context-Aware Cross-Site Scripting Filter Evasion*; Department of Computer, Control, and Management Engineering Antonio Ruberti: Rome, Italy, 2012.
52. Van Rooij, O.; Charalambous, M.A.; Kaizer, D.; Papaevripides, M.; Athanasopoulos, E. webFuzz: Grey-Box Fuzzing for Web Applications. In *Computer Security–ESORICS 2021*; Bertino, E., Shulman, H., Waidner, M., Eds.; Lecture Notes in Computer Science; Springer International Publishing: Cham, Switzerland, 2021; Volume 12972, pp. 152–172. [CrossRef]
53. Huang, J.; Zhang, J.; Liu, J.; Li, C.; Dai, R. UFuzzer: Lightweight Detection of PHP-Based Unrestricted File Upload Vulnerabilities Via Static-Fuzzing Co-Analysis. In Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses, San Sebastian, Spain, 6–8 October 2021; ACM: San Sebastian, Spain, 2021; pp. 78–90. [CrossRef]
54. Lee, T.; Wi, S.; Lee, S.; Son, S. FUSE: Finding File Upload Bugs via Penetration Testing. In Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2020,San Diego, CA, USA, 23–26 February 2020.
55. Benedikt, M.; Freire, J.; Godefroid, P. VeriWeb: Automatically Testing Dynamic Web Sites. In Proceedings of the 11th International World Wide Web Conference(WWW'2002), Honolulu, HI, USA, 7–11 May 2002.
56. Huang, Y.W.; Huang, S.K.; Lin, T.P.; Tsai, C.H. Web Application Security Assessment by Fault Injection and Behavior Monitoring. In Proceedings of the 12th International Conference on World Wide Web, Budapest, Hungary, 20–24 May 2003; Association for Computing Machinery: New York, NY, USA, 2003; pp. 148–159. [CrossRef]
57. Saxena, P.; Hanna, S.; Poosankam, P.; Song, D. FLAX: Systematic Discovery of Client-side Validation Vulnerabilities in Rich Web Applications. In Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 28 February–3 March 2010.