FUSE: Finding File Upload Bugs via Penetration Testing

Taekjin Lee*[†‡], Seongil Wi*[†], Suyoung Lee[†], Sooel Son[†]

[†]School of Computing, KAIST
[‡]The Affiliated Institute of ETRI

***Abstract*—An Unrestricted File Upload (UFU) vulnerability is a critical security threat that enables an adversary to upload her choice of a forged file to a target web server. This bug evolves into an Unrestricted Executable File Upload (UEFU) vulnerability when the adversary is able to conduct remote code execution of the uploaded file via triggering its URL. We design and implement FUSE, a penetration testing tool designed to discover UFU and UEFU vulnerabilities in server-side PHP web applications. The goal of FUSE is to generate upload requests; each request becomes an exploit payload that triggers a UFU or UEFU vulnerability. However, this approach entails two technical challenges: (1) it should generate an upload request that bypasses all content-filtering checks present in a target web application; and (2) it should preserve the execution semantic of the resulting uploaded file. We address these technical challenges by mutating standard upload requests with carefully designed mutations that enable the bypassing of content-filtering checks and do not tamper with the execution of uploaded files. FUSE discovered 30 previously unreported UEFU vulnerabilities, including 15 CVEs from 33 real-world web applications, thereby demonstrating its efficacy in finding code execution bugs via file uploads.**

I. Introduction

Sharing user-provided content has become a *de facto* standard feature of modern web applications. Facebook, Instagram, and Twitter have increasingly invited users to upload their own pictures, videos, and text posts. A content management system (CMS) is another representative web application supporting file uploads. The WordPress [23] and Joomla [10] platforms, accounting for a combined 65% of CMS market share [20], enable users to upload their images, PDFs, and TAR files. This upload functionality is a prevalent feature that server-side web applications support.

Meanwhile, the upload feature poses a security risk wherein an attacker can upload her arbitrary file to a target server and exploit it as a stepping-stone to further opportunities for compromising the target system. Therefore, it is essential for web application developers to prevent an attacker from abusing this upload functionality. A widespread practice for its mitigation is to implement content-filtering checks that disable the uploading of specified file types that pose a critical security risk. For example, WordPress forbids its users from uploading any PHP files because an adversary could execute an uploaded PHP file that allows unrestricted access to internal server resources.

Unrestricted File Upload (UFU) [18] is a vulnerability that exploits bugs in content-filtering checks in a server-side web application. An adversary, called an *upload attacker*, leverages her limited privilege to upload a malformed file by exploiting a UFU vulnerability. The successful uploading of a forged file poses a *potential code execution* risk [18]. A system administrator may accidentally run this forged but still executable file while vetting the new file, or a bug in an existing software can facilitate the execution of the uploaded file.

This UFU vulnerability becomes even more critical when the adversary is able to trigger code execution of an uploaded file via its URL; this means that the adversary is capable of conducting *arbitrary code execution* by invoking the URL. We refer to a bug in content-filtering checks as an Unrestricted Executable File Upload (UEFU) vulnerability when (1) it allows the upload of an executable file and (2) the adversary is able to remotely run this executable file on a target web server or a victim's browser by invoking a URL.

There have been previous studies on detecting various web vulnerabilities. Several techniques have been used in attempts to detect taint-style vulnerabilities, including XSS and SQLI, via static analyses [43, 49, 63, 65] or dynamic executions [29, 53]. Conducting symbolic execution has also been explored for finding logic bugs [59, 62] and generating attack exploits [26]. However, few research studies have addressed finding U(E)FU vulnerabilities [40].

Contributions. In this paper, we propose FUSE, a penetration testing system designed to identify U(E)FU vulnerabilities. Penetration testing is a widely practiced testing strategy, especially in finding security bugs [32, 44, 48, 51]. One invaluable advantage of penetration testing is that it produces actual exploits that trigger inherent vulnerabilities. Each reported exploit payload helps an auditor better understand system weaknesses and assures their lower bound security level.

The effectiveness of penetration testing solely depends on generating inputs likely to trigger U(E)FU vulnerabilities, which entails two technical challenges: (1) FUSE should generate an upload request that bypasses application-specific content-filtering checks present in the target web application, resulting in a successful upload; and (2) a successful upload request should drop a file that the target web server or a browser is able to execute.

To address these challenges, we propose a novel mutation-based algorithm for generating upload requests that elicit U(E)FU vulnerabilities. FUSE begins by generating four *seed* upload requests; each request attempts to upload either a PHP,

*Both authors contributed equally to the paper

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23126
www.ndss-symposium.org

HTML, XHTML, or JS file. The target application may block these seed requests because each one attempts to upload an executable file, which is inadmissible to the target application.

FUSE, therefore, mutates each seed request by applying combinations of 13 carefully designed mutation operations. We designed each one to help bypass content-filtering checks as well as preserve the execution semantic of the seed file. Specifically, we defined five objectives that trigger common mistakes in implementing content-filtering checks. We then implemented concrete mutation methods, each of which achieves at least one objective, thereby addressing the first challenge. At the same time, these mutation operations do not tamper with constraints required for the seed file to be executable by a target execution environment, thus preserving the execution semantic of the seed file. That is, these mutations are key components addressing the aforementioned two technical challenges. FUSE then sends these generated requests to a target PHP application in the attempt to upload mutated variants of seed files. Finally, it checks whether the uploaded file is executable by accessing its URL, which is computed from a given configuration file or obtained from the file event monitoring tool at a target server.

We evaluated FUSE on 33 popular real-world web applications; FUSE discovered 30 new UEFU vulnerabilities with corresponding upload requests that caused arbitrary code execution. These uploading requests are valuable test inputs that trigger inherent vulnerabilities, thereby helping developers understand their root causes. We reported all findings to the corresponding vendors and received 15 CVEs.

In summary, this paper demonstrates that it is feasible to conduct an effective penetration testing with carefully designed mutation operations that do not tamper with code execution of seed files but are nevertheless effective in bypassing content-filtering checks. Because our mutation-based testing strategy is compatible with off-the-shelf penetration testing tools [3, 32, 44] for finding web vulnerabilities, FUSE is able to contribute to those testing tools extending to cover U(E)FU vulnerabilities. To support open science and further research, we will release FUSE at https://github.com/WSP-LAB/FUSE.

II. Background

We explain a general procedure for uploading files in PHP server-side web applications. We then describe UFU and UEFU vulnerabilities, as well as their security impacts.

A. File Upload in PHP Web Applications

PHP is a popular server-side web programming language. Approximately 80% of web servers among the Alexa top 10 million sites use PHP to implement various services, including CMS and social forums [21].

Upload functionality is a key feature that PHP supports. A common upload procedure begins with a client browser sending an HTTP(S) multipart request [50], originating from an HTML form. This request is usually sent via POST, which embodies the user's selection of a local file.

The recipient PHP interpreter of this upload request extracts the file and then moves this file to a temporary directory, as specified in php.ini. Finally, the PHP application, invoked by this upload request, conducts content-filtering checks that determine whether the uploaded file conforms to the developers' expectations.

```
1  <?php
2    $black_list = array('js','c','php3',...,'php7')
3    if (!in_array(ext($file_name), $black_list)) {
4      $file_path = $base_path . sanitize($file_name)
5      $uploaded = move($tmp_file_path, $file_path);
6    }
7    else {
8      message('Error: forbidden file type');
9    }
10 ?>
```

Fig. 1: Example snippet of content-filtering checks implemented in Monstra.

Figure 1 shows an example of content-filtering checks in the Monstra CMS application. Line (Ln) 3 extracts the extension from the uploaded file name via the ext function and then checks whether it is among the blacklisted extensions hardcoded at Ln 2. It forbids uploading any file with a blacklisted extension that poses a potential security threat. Lastly, Ln 5 moves the uploaded file from the temporary directory to the sanitized file path, which Ln 4 computes to specify the upload directory where uploaded files should be stored.

B. UFU and UEFU Vulnerabilities

A UFU vulnerability [18] is a security bug that allows an adversary to upload arbitrary files that developers do not expect to accept. This vulnerability stems from a flawed implementation of content-filtering checks, which are designed to accept only admissible files. For instance, Monstra does not accept JS files from their clients as Ln 2 in Figure 1 shows, which poses a potential security threat. However, assume that one uploads the bypass.Js file, which contains arbitrary JS code. This file triggers a UFU vulnerability because it bypasses the content-filter check in Ln 3, which checks the file extension in a case-sensitive manner. This uploaded JS file imposes a risk of potential code execution (PCE). An upload attacker is able to abuse hosts running the vulnerable Monstra to distribute a malicious JS script that could run on the victims' browsers.

In this paper, we define an Unrestricted Executable File Upload (UEFU) vulnerability as a UFU vulnerability that allows arbitrary code execution (CE) via a URL leading to an uploaded executable file. Thus, UEFU vulnerabilities are a subset of UFU vulnerabilities. We only consider UFU and UEFU vulnerabilities that allow the upload of a file, executable by a PHP interpreter or a browser. Specifically, we focus on identifying U(E)FU vulnerabilities that enable the uploading of four file types: PHP, HTML, XHTML, and JS. Each file type requires different conditions to cause PCE or CE.

PHP. When (1) an upload attacker is able to upload a PHP file by exploiting a UFU vulnerability and (2) the attacker is capable of executing this uploaded file via a publicly accessible URL, the exploited UFU vulnerability becomes a UEFU vulnerability, which results in remote CE. For example, consider the adversary successfully uploading the simple PHP web shell code in Figure 2a. Since the uploaded script can be invoked via a URL with any crafted parameters, an adversary

```
<?php
  system($_GET['c']);
?>
```

(a) Uploaded PHP file.

```
<html>
  <script>
    alert('xss');
  </script>
</html>
```

(b) Uploaded (X)HTML file.

Fig. 2: Examples of uploaded attack files.

is capable of executing any system commands. This poses a critical threat such that the adversary is able to access local file resources and databases [4], inject shell commands and scripts [37], and conduct Server-Side Request Forgery (SSRF) attacks [56].

`.htaccess` is an Apache configuration file that contains configuration directives on a per-directory basis. It often defines an access-control policy on files under the directory where the file is located as well as determines which file extensions should be run by a PHP interpreter [1]. PHP application developers may limit the entry points that allow access to uploaded files by implementing `.htaccess`. In this case, the attacker is unable to invoke the uploaded PHP file via a URL. However, it still qualifies as a UFU vulnerability causing PCE. Consider a vulnerable web application with a Local File Inclusion (LFI) [16] that allows an attacker to embed any server-side file for the execution of this application, as the following example shows.

```
<?php include($_GET['page']); ?>
```

Regardless of the existence of a publicly accessible URL, a web attacker is capable of executing an uploaded PHP by leveraging this LFI vulnerability.

(X)HTML. An uploaded HTML or XHTML file is also a critical attack vector for injecting malicious JS code, thus imposing a CE threat. Assume that the adversary takes on the role of a web attacker [31] by uploading the HTML file shown in Figure 2b and lures victims into visiting the URL that leads to the uploaded file. The adversary is thus able to trigger the execution of malicious JS scripts with the vulnerable web server origin on the behalf of a victim. This allows unrestricted access to sensitive information in the victim's cookies and local storage governed by the Same Origin Policy (SOP) [57]. By definition, this is a stored cross-site scripting attack [17]. Any domain-based Content Security Policy (CSP) [36, 54, 60, 61] provides little to no protection because the URL rendering the malicious HTML file is within the target web server domain.

JS. A UFU vulnerability that allows the upload of a JS file imposes a PCE threat. Many network-level firewalls or CSPs use domain names to block content resource requests fetching JS files via blacklists or whitelists. By uploading malicious JS scripts to a vulnerable web server, the adversary can distribute those malicious JS scripts to victims or bypasses CSPs that list this vulnerable web server as trustworthy [47, 64].

III. Motivation

This section describes a threat model and the attacker's concrete capabilities of exploiting U(E)FU vulnerabilities that FUSE is designed to find. It then depicts two technical challenges to systematically find U(E)FU vulnerabilities and summarizes our approach to tackling these challenges.

```
1  <?php
2  function check_filetype_and_ext
3   ($file, $filename, $mimes) {
4   // Infer a type from filename.
5   $filetype = check_filetype($filename, $mimes);
6   ...
7   if($type && !$real_mime &&
   ↪  extension_loaded('fileinfo')) {
8     $finfo = finfo_open(FILEINFO_MIME_TYPE);
9     $real_mime = finfo_file($finfo, $file);
10     // Check an inferred MIME type.
11     if(!in_array($real_mime,
    ↪  array('application/octet-stream',..),true)){
12       $type = $ext = false;
13     }  }
14     ...
15     $allowed = get_allowed_mime_types();
16     if (!in_array($type, $allowed)){
17       $type = $ext = false;
18  }  }  }  }
19  ?>
```

Fig. 3: Simplified content-filtering logic in WordPress.

A. Threat Model

We assume an *upload attacker*. The attacker has a limited privilege of uploading legitimate files granted by a target web application; she is unable to use any other system-level upload channels, such as the secure file transfer (SFTP) or the secure copy protocol (SCP). For instance, an upload attacker could be a registered user of a WordPress website. The adversary can perform only limited operations according to her access control role as the developer intended. That is, she cannot upload any files using measures other than the emplaced upload functionality that WordPress provides. The goal of the adversary is to upload a file that initiates CE at a server-side PHP interpreter or a client-side browser and to subsequently trigger the execution of the uploaded file via a publicly accessible URL. The adversary may initiate the execution of an uploaded file by leveraging an existing LFI vulnerability [16].

B. Technical Challenges

Finding UEFU vulnerabilities entails two technical challenges: (1) identifying bugs in application-specific content-filtering checks, and (2) confirming whether such bugs allow the successful upload of a file executable by a PHP interpreter or a browser.

Application-specific checks. Different applications implement their own content-filtering checks in idiosyncratic ways. Figures 1 and 3 show two different types of content-filtering logic. The content-filtering logic of Monstra only checks whether a user-provided file extension conforms to a predefined extension blacklist. On the other hand, the content-filtering logic of WordPress in Figure 3 begins by extracting the extension from a file name at Ln 5. When the given file is an image, the omitted logic at Ln 6 infers its MIME type, computing `real_mime`. If `real_mime` is not determined, Ln 9 infers its MIME type from a given file based on its content by invoking the `finfo_file` built-in function. Ln 16 finally checks whether the given file is admissible by leveraging two inferred MIME types from its file extension and content.

Even worse, several applications implement content-filtering logic across different places in their applications, thereby making it difficult to understand their underlying logic even with manual analyses. In our benchmark consisting of 33 popular CMSs, *we observed that no application implemented the identical content-filtering logic.*

This engineering practice entails a technical challenge for identifying bugs in such application-specific content-filtering checks. The majority of web applications are accompanied by neither specifications of their admissible files nor annotations indicating whether the checks are located. Identifying or inferring such specifications to begin any procedure is a challenging problem [34, 49, 58]. Furthermore, the identification of content-filtering checks is not enough. It is essential to find test inputs that would bypass such checks but deviate from developers' expectations, thus triggering bugs.

Symbolic execution is certainly applicable to systematically finding bugs in content-filtering checks [26, 27, 39, 40, 45, 59, 62, 65]. However, the aforementioned engineering practice becomes problematic when conducting symbolic execution. By nature, symbolic execution requires the specifications that pinpoint exact code locations after bypassing content-filtering checks. This requirement demands a deep understanding of a target application, thus potentially hindering its application by auditors with less domain knowledge who want to test diverse applications.

Executable uploaded files. There is another technical challenge; a bug in content-filtering checks should allow the successful upload of a file that a target web server or a browser can execute. Addressing this challenge involves answering the research question: *What constraints should be preserved in an uploaded file such that it is executable by a web server or a browser?* Identifying such constraints requires a deep understanding of web server and browser behaviors for executing a given file.

Our methodology. We focus on finding U(E)FU vulnerabilities that allow code execution of uploaded seed files that PHP interpreters with Apache or three major browsers (i.e., Chrome, Firefox, and Internet Explorer) execute. To this end, we propose a penetration testing system to address the aforementioned two technical challenges.

To address the first challenge, we propose eliciting unintended erroneous behaviors by providing forged upload requests that are likely to trigger inherent bugs while avoiding to generating specifications for the intended semantics of application-specific checks. In particular, when generating upload requests, we apply carefully designed mutation operations that help bypass buggy application-specific checks, whose root causes stem from common mistakes of developers.

We also analyze the source code of Chrome, Firefox, Apache, and PHP engines to identify the constraints required for executable files. When generating upload requests, we ensure that these identified constraints are preserved in attack files in the upload requests, which addresses the second challenge. Also, in Section VII-D, we demonstrate that the changes made to these constraints due to software updates are so few that the execution of most mutation variants remains consistent across different versions of Chrome, Firefox, Safari, and the PHP engines.

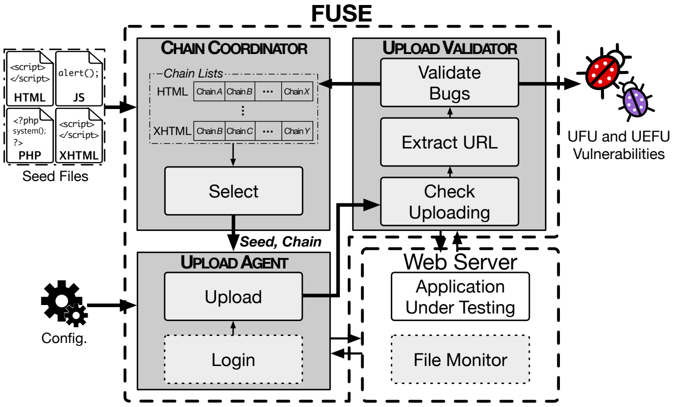


Fig. 4: Overview of FUSE architecture.

IV. Overview

FUSE takes in a set of seed files and a configuration file given a target server-side web application. FUSE then initiates a penetration testing campaign. During the campaign, FUSE mutates the upload request of seed files by applying the combinations of 13 carefully designed mutation operations, and attempts the uploads of mutated seed files by sending those requests. Once the campaign is over, FUSE reports functional upload requests that demonstrate the presence of U(E)FU vulnerabilities.

Figure 4 illustrates the overall architecture of FUSE, which consists of three components: Chain Coordinator, Upload Agent, and Upload Validator. At a high level, these components work in tandem to perform three steps; (1) the Chain Coordinator prepares a testing strategy for each of the four seed files; (2) the Upload Agent builds upload requests, mutates those requests according to the testing strategy, and sends those mutated requests in an attempt to upload variants of the seed files; and (3) the Upload Validator checks whether the uploaded files are accessible and executable via publicly accessible URLs.

Chain Coordinator. The Chain Coordinator constructs a testing strategy, called a *chain list*. It specifies how to generate a series of mutated upload requests. Each *chain* in this chain list entails a list of mutation operations that FUSE applies to a seed upload request. Each mutated upload request is thus a computation result of applying mutations in a chain to a seed upload request.

Upload Agent. This module is responsible for generating an upload request for a given seed file and mutating the original request according to a given chain computed by the Chain Coordinator. A target application often requires completing an authentication procedure and sending a valid CSRF token with each attempted upload request. Therefore, the Upload Agent addresses the authentication procedure and appends valid CSRF tokens to facilitate the upload procedures.

Upload Validator. The Upload Validator checks whether generated requests succeed in uploading files and obtains the publicly available URLs of these uploaded files. By accessing these files through the computed URLs, the Upload Validator checks whether the uploaded files are executable.

```
1  /* Required Parameters */
2  login_page    = [Login page URL.],
3  credential    = {
4     id         = [Username.],
5     pw         = [Password.]},
6  upload_page   = [Uploading page URL.],
7  token_re      = [Regex for matching a CSRF token.],
8  /* Optional Parameters */
9  success_re    = [Regex for a successful upload.],
10 response_re   = [Regex for file URLs.],
11 url_prefix    = [Common prefix of file URLs.],
```

Fig. 5: Simplified FUSE configuration template file.

V. Design

Given a configuration file, FUSE conducts three phases. Phase I computes a testing strategy, which we refer to as a chain list, for each seed file. Phase II executes this testing strategy by constructing a seed request for each seed file, mutating these seed requests according to the chain list, and sending mutated requests. Phase III obtains the accessible URLs leading to successfully uploaded files and checks the execution capability of these uploaded files.

A. Specifying a Testing Campaign

FUSE takes in two inputs: a set of seed files and a configuration file. Each seed file becomes a source for building a standard upload request, which is called a *seed request*. FUSE also uses a user-provided configuration file that specifies parameters for a target PHP application.

Figure 5 shows a configuration template. It specifies authentication credentials, URLs for the login and upload webpages, and CSRF token fields from which FUSE extracts tokens. The parameters in Lines (Lns) 9-11 are optional as some applications may not require them. They specify how to obtain the URLs for uploaded files. Section V-D explains how FUSE utilizes each parameter in detail.

We argue that specifying this configuration file is an acceptable cost for finding U(E)FU vulnerabilities. Widespread web penetration testing tools require comparable configuration effort. SQLmap [14] requires auditors to specify login cookie credentials, target URLs, and parameters to inject payloads. Arachni [3] and Burp [5] crawl target URLs and injection parameters by default but still demand the same information for better coverage and precise scanning. Zap [25] takes advantage of its network proxy tools to generate sitemaps and specify attack targets via user interactions, thus systematically generating such configuration information.

The additional configuration cost for FUSE is necessary to define the *success_re*, *response_re*, and *url_prefix* parameters. The *success_re* parameter indicates whether an upload attempt is successful. The *response_re* and *url_prefix* parameters are for computing the URLs leading to uploaded files. These parameters can be omitted when leveraging the File Monitor at a target web server (§V-D). However, these parameters exist to support testing scenarios in which placing the File Monitor is not a viable option.

B. Phase I: Chain Coordination

This chain coordination step generates a testing strategy specifying how to mutate a given seed request. Recall that a *chain* is a list of mutation operations. This testing strategy entails a list of chains, which we call a *chain list*. The goal of this chain coordination is to exhaustively explore all feasible mutation combinations, thus contributing to FUSE generating diverse upload requests and finding new bugs. Note that each mutation operation is designed to bypass one kind of content-filtering check. Therefore, the combination of those certainly increases the odds of bypassing multiple content-filtering checks. Our evaluation demonstrates that there exist numerous bugs that FUSE could miss without considering mutation combinations (§VII-D).

The Chain Coordinator begins by creating an initial chain list for each seed request. For each seed request, it permutes all applicable mutation operations and then orders them by chain length. For instance, if the mutation operations applicable to the HTML seed are M1, M2, and M3, the chain list is as follows.

HTML: {∅, M1, M2, M3, M1M2, M1M3, M2M3, M1M2M3}

It is possible for two different mutation operations to conflict with each other in the case that they revise the overlapping portions of a seed request. The Chain Coordinator removes such spurious chains to purge unnecessary mutations. For example, if M1 conflicts with M2, the revised chain list for the previous example becomes the following.

HTML: {∅, M1, M2, M3, ~~M1M2~~, M1M3, M2M3, ~~M1M2M3~~}

Another functionality of the Chain Coordinator is to remove chains based on a previous upload attempt result obtained from Phase III to conduct an efficient penetration testing campaign. If a chain contributes to a successful upload, the Chain Coordinator purges all other chains that include the successful chain. Because the chain list is ordered according to its chain length, FUSE always picks a short chain rather than other longer chains that include this short chain. Our purpose is to report distinct minimum-length chains for successful exploits. For example, if the chain M1 successfully triggers a UFU vulnerability, the Chain Coordinator removes all other chains that include M1 from the chain list as follows:

HTML: {$\underbrace{\emptyset, \mathtt{M1}}_{tested}$, M2, M3, ~~M1M3~~, M2M3}

Also, if the chain of ∅ (i.e., no mutation to the seed request) triggers a UFU vulnerability, the Chain Coordinator removes all chains in the chain list. In other words, when a seed request succeeds in uploading its seed file, FUSE sends no further mutated upload requests originating from this seed request. When a target application implements no measure to prevent UFU vulnerabilities, finding diverse test cases becomes pointless. On the other hand, if a seed request fails, it indicates the existence of content-filtering checks against which FUSE performs a penetration testing campaign.

C. Phase II: Mutating and Sending Upload Requests

The Upload Agent starts by performing the authentication procedure of a target web application. It leverages the login_page and credential parameters from a given

Algorithm 1: File Upload Algorithm.

```
1  function Upload(conf, seed, chain)
2      unique ← RandStr(32)
3      url ← conf.upload_page
4      $token_u$ ← ExtractTokens(url, conf.token_re)
5      request ← ConstructRequest(url, $token_u$, seed)
6      for m ∈ chain do
7          request ← MutateRequest(request, m)
8      request ← PostProcess (request, unique)
9      response ← SendRequest (url, request)
10     return request, response, unique
```

configuration file to construct an authentication request and sends this request to complete the authentication procedure. The Upload Agent then generates upload requests by mutating the seed request and sends these requests. Algorithm 1 describes this uploading procedure. It obtains a given configuration file (*conf*), seed file (*seed*), and *chain*. It first assigns a unique identifier in Ln 2, which is a reference index used for the later validation process.

Because a target web application often requires a valid CSRF token, the `ExtractTokens` function dynamically extracts a CSRF token from an upload page. It internally fetches the upload webpage and extracts the form element corresponding to a CSRF token by leveraging a regular expression specified in *conf*. The `ConstructRequest` function in Ln 5 then constructs a seed request that attempts to upload the *seed* by adding the extracted CSRF token. The Upload Agent then mutates *seed* by applying each mutation in *chain*, as Lns 6-7 show. Ln 8 performs the post-processing of the mutated upload request to facilitate the later validation process. Specifically, it changes the upload file name, assigning it a *unique* value and appends this value in the comment portion of the file to be uploaded. Finally, the `SendRequest` function in Ln 9 sends the mutated and post-processed *request* to the target *url* and returns the response received from the target application.

D. Phase III: Upload Validation

The Upload Validator performs three tasks: (1) it checks whether each attempted upload request successfully drops a file at the web server hosting the target application; (2) it computes the URL leading to the uploaded file; and (3) it confirms whether this obtained URL invokes the execution of the uploaded file.

As the first task in vetting a successful upload, the Upload Validator checks whether the *response* to an upload request is free from any error messages by default. The Upload Validator leverages a regular expression (`success_re`) defined in the configuration file that checks for the existence of a pattern in the response indicating a successful upload.

For the second task, the Upload Validator has three different methods of obtaining the URL of an uploaded file. Because various applications differ in assigning URLs to uploaded files, we generalize those into three methods. We explain these methods from the simplest approach to the most sophisticated one.

Common prefix of URLs. The Upload Validator uses a user-provided parameter, `url_prefix`, which indicates the common prefix of URLs leading to all the uploaded files. If this parameter is set, the Upload Validator simply concatenates the URL value extracted with `url_prefix` and an upload file name, thus generating the final URL.

Upload response and summary webpage. Several applications, including HotCRP, present the URL of an uploaded file in the response to its upload request. The Upload Validator leverages a user-provided parameter, `response_re`, to extract the URL leading to this uploaded file.

Instead of checking the upload response, the Upload Validator is able to reference a specified summary page listing all accessible URLs leading to uploaded files. The Upload Validator leverages the *unique* identifier from Algorithm 1. Each URL already has a *unique* identifier in its file path, and the fetched content from this URL contains a *unique* identifier in its body. Thus, the Upload Validator is able to map each URL to an upload request by leveraging the *unique* identifier as a joining key.

File Monitor. The previous two methods are highly dependent on user-provided parameters and the understanding of a target application. Furthermore, several applications use random file names for their uploaded files and provide no summary page, which makes defining the `url_prefix` and `response_re` parameters infeasible. To handle such cases, the Upload Validator uses the File Monitor. The File Monitor is a monitoring component that is installed at the web server hosting a target web application. It is a one-time setup tool that monitors any file creation event under a web root directory. [1]

For each creation event, the File Monitor stores the absolute path of the created file and the MD5 hash value of its content. When the Upload Validator sends the hash value to retrieve the URL leading to a successfully uploaded file, the File Monitor responds with the stored absolute path that matches the hash value of the file. The Upload Validator computes the URL from the received absolute path by replacing the web root directory with the web server domain name.

Finally, the Upload Validator validates whether each obtained URL indeed invokes the execution of an uploaded file, which could be different from its seed file. For the PHP seed file, we implemented the code that dynamically generates '`FUSE_GEN`.' The Upload Validator invokes a mutated version of this seed via its URL and checks whether the response page contains '`FUSE_GEN`', which demonstrates the successful CE of the PHP variant. Otherwise, the Upload Validator considers such cases as PCE risks.

For an uploaded HTML, JS, or XHTML file, the Upload Validator checks the difference between the attempted upload file contained in an upload request and the uploaded file fetched from the obtained URL. If there is no difference, those uploaded files are highly likely to be executable because none of the applied mutations tampers with the execution of the mutated file. Next, the Upload Validator checks whether the `Content-Type` header in the response is among our selections of 10 MIME types. Recall that JS, HTML, and XHTML files are executed at client-side browsers, and these browsers reference the MIME type in the `Content-Type` header to decide whether the fetched content is executable.

[1] We used the default Apache web root directory

We empirically collected the aforementioned 10 MIME types. For each JS or (X)HTML seed file, we fetched the MIME types while varying the `Content-Type` header values and checked whether they were indeed executable in Chrome, Firefox, or Internet Explorer headless browsers.

E. Uploading `.htaccess`

FUSE further checks the feasibility of uploading a `.htaccess` file. If an upload attacker is able to control a `.htaccess` file, she is able to invoke a PHP interpreter to execute an uploaded file with any extension as well as to make this uploaded file accessible. This is a critical security threat that enables a UFU vulnerability, which imposes a PCE risk, to evolve into a UEFU vulnerability that results in CE.

Specifically, after completing Phase III, FUSE attempts to upload an arbitrary `.htaccess` file. We programmed this `.htaccess` file to allow arbitrary extensions to be executed by a PHP interpreter. To check whether the `.htaccess` file has successfully uploaded, FUSE uploads another arbitrary image file with metadata that embeds in the PHP seed file. It then validates the execution of the uploaded image file via a PHP interpreter by invoking the URL leading to this image file.

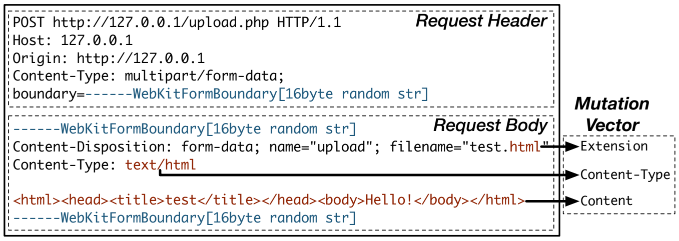
VI. Mutation Operations

The main goal of the mutations is to transform a given upload request in a way that its resulting upload file preserves the execution semantic of its seed file and the mutated request is likely to bypass content-filtering logic. To achieve this goal, we started by identifying mutation vectors that an upload attacker is able to manipulate. Assuming an upload attacker who exploits these mutation vectors, we conducted a preliminary study to identify common developer mistakes in performing content-filtering checks.

Preliminary study. We investigated known CVEs, existing evasion techniques from the Internet, and previous studies [30, 41]. We also examined what built-in methods that mature applications leverage for content-filtering checks in nine popular applications, including WordPress and Joomla. Based on these investigations, we generalized the existing attack techniques into five objectives that exploit different types of developer mistakes. We then designed 13 operations, each of which instantiates one or two of the defined objectives, thereby triggering inherent mistakes and bypassing emplaced content-filtering logic. Note that five of 13 operations (i.e., M5, M7, M9, M10, and M13) are proposed by our work.

Execution constraints. When designing each mutation, we adjusted the operation to preserve the execution semantic of a seed file. We investigated basic constraints for a given file that an Apache web server or a browser requires for its execution. We manually analyzed the source code of Chrome 74, Firefox 68, eight different versions of Apache `mod_php` modules, and PHP 5.6 interpreter engines to understand which constraints should be preserved for the seed files to be executable.

We observed that a PHP interpreter executes a PHP file that contains the PHP start tag (i.e., `<?php` or `<?`). However, this invocation of a PHP interpreter is governed by an Apache `mod_php` module. This module requires an executable PHP file to have one of the seven PHP-style file extensions (e.g., `php3, phar`) for its execution via direct URL invocations. In the Chrome and Firefox browsers, we also identified that an executable HTML file must start with pre-defined start tags within its first 512 bytes with subsequent valid HTML code, which is well aligned with the models that Barth *et al.* extracted [31]. An executable XHTML file shares the same constraints as the HTML case but requires the presence of `xmlns` tags. We also investigated other browser-supported file types (i.e., SVG and EML) that allow embodying JS scripts. When implementing each mutation operation, we ensured that they reflected these constraints, thus preventing the mutation from tampering with these constraints.



```
POST http://127.0.0.1/upload.php HTTP/1.1
Host: 127.0.0.1
Origin: http://127.0.0.1
Content-Type: multipart/form-data;
boundary=------WebKitFormBoundary[16byte random str]

------WebKitFormBoundary[16byte random str]
Content-Disposition: form-data; name="upload"; filename="test.html"
Content-Type: text/html

<html><head><title>test</title></head><body>Hello!</body></html>
------WebKitFormBoundary[16byte random str]
```

Fig. 6: Message structure of an HTTP multipart request and mutation vectors.

Mutation vectors. FUSE mutates the fields of an HTTP(S) multipart request [50], which is generally constructed by clients to upload files and data to a web server. Figure 6 represents the standard message format of an HTTP multipart request. In the *request body* of the upload request (Figure 6), FUSE considers three mutation vectors to modify its corresponding field: (1) `Extension`, (2) `Content-Type`, and (3) `Content`. From the point of view of the file, each vector is represented as follows.

- `Extension`: the extension of a file name.
- `Content-Type`: the MIME [38] type of a file.
- `Content`: the binary content or plain text of a file.

Mutation objectives. The followings enlist five key objectives derived from the aforementioned preliminary study.

1) *Checking the absence of content-filtering checks*: We observed that several applications do not perform any checks on incoming upload requests. FUSE achieves this objective by sending the seed request for each executable seed file without applying any mutation.

2) *Eliciting incorrect type inferences based on* `Content`: This goal is inspired by the previous approaches that generate a file with an inferred type that varies between different execution environments [30, 41]. They demonstrated chameleon attacks, which disguise one type of file as another type to evade the type inference heuristics of malware detectors or browsers. We extend this idea to induce different views on an uploaded file type between web applications and the execution environment where the uploaded file runs. Specifically, we aimed to cause erroneous type inferences from PHP built-in functions, including `finfo_file` and `mime_content_type`, which references the `Content` part.

3) *Exploiting incomplete whitelists or blacklists based on* `Extension`: We observed that different applications differ in specifying prohibited extensions (blacklist) or allowed extensions (whitelist). The goal is to exploit this inconsistency of

OP	Description	Seed File(s)	Objectives
M1	Prepending a resource header	PHP, HTML	2
M2	Inserting a seed into metadata	PHP, HTML, JS	2
M3	Changing the content-type of a request	PHP, HTML, XHTML, JS	5
M4	Changing a file extension	PHP, HTML, XHTML, JS	3
M5	Replacing PHP tags with short tags	PHP	4
M6	Converting HTML into EML	HTML, XHTML	2, 3
M7	Removing a file extension	PHP, HTML, XHTML, JS	3
M8	Converting a file in SVG	HTML	3
M9	Prepending an HTML comment	HTML, XHTML	2, 4
M10	Changing a file extension to an arbitrary string	PHP, HTML, XHTML, JS	3
M11	Converting a file extension to uppercase	PHP, HTML, XHTML, JS	3
M12	Prepending a file extension	PHP, HTML, XHTML, JS	3
M13	Appending a resource header	PHP, HTML, XHTML, JS	2

TABLE I: List of mutation operations for each seed file.

whitelists or blacklists of extensions, which presents opportunities to allow the uploads of impermissible files.

4) *Bypassing keyword filtering logic based on* `Content`: The goal is to bypass the filtering logic of applications that search for certain program-specific keywords, including `<?php`, `<html>`, and `<script>`, to infer an uploaded file type.

5) *Bypassing filtering logic based on* `Content-Type`: We observed that several applications often accept the MIME type specified in the `Content-Type` without checking the actual type of the file in the `Content`. The goal is to inject incorrect MIME types to bypass content-filtering checks.

Table I summarizes the list of mutation operations that we designed to address the five objectives above. Note that achieving the first objective demands no mutation because accepting seed requests with no mutation implies the absence of content-filtering checks. Each mutation addresses at least one objective and corresponds to certain seed files. For example, M1 is designed to achieve the second objective and is only applicable to two seed request types that upload HTML or PHP seed files.

Mutation conflicts. Recall from Section V-B that we predefined a set of conflicting mutation operations for each mutation operation and excluded such conflicting operations when creating the chain list. For a given operation ($M1$), we defined a conflicting mutation ($M2$) as when (1) both $M1$ and $M2$ revise the same portion of a mutation vector, or (2) $M1$ combined with $M2$ causes a CE failure, thus rendering $M2$ unnecessary. When enumerating the permutations of the 13 mutations set, we discarded a combination in which one of its mutation operations conflicted with other mutation operations.

M1: Prepending a resource header. M1 prepends the 1024 bytes from the headers of six resource files (GIF, JPG, PDF, PNG, TAR_GZ, and ZIP) to the `Content` of a given upload request. Thus, applying M1 means generating a mutation request for each resource file, thus generating six distinct mutation requests. The intention is to deceive the type inference heuristics of a target PHP application. A common method to filter out a malicious file is to infer its type and to reject the file based on the inferred type. We observed that several PHP applications inferred a file type by matching a prepared signature to the header part of a file. This observation led us to define the M1 operation. M1 is applicable to PHP and HTML seed requests; however, M1 is not applicable to JS and XHTML because no browser is able to execute a JS or XHTML file with a resource file header.

M2: Inserting a seed into resource metadata. M2 injects the `Content` of a given upload request into the metadata portion of six resource files (JPG, PNG, GIF, ZIP, PDF, and BMP), thus generating six distinct mutations. FUSE analyzes the structure of each resource file and identifies the specific chunk blocks that include comment metadata. Thereafter, our system injects an upload file into that block as comment metadata. Finally, FUSE changes the `Content` in the seed request with the corresponding values of the modified resource file. For instance, FUSE injects a PHP seed into the comment part of the GIF89a metadata. Unlike to M1, which tries to upload an incomplete resource file, M2 uploads a complete resource file so that most image viewers render its thumbnail and image without any error. We checked that the M2 operation does not tamper with code execution of a PHP and an HTML file.

M3: Changing the Content-Type of an upload request. M3 changes the `Content-Type` of an upload request into one MIME [38] type of the six resource files (JPG, PNG, GIF, TAR_GZ, ZIP, and PDF). We observed that some applications leverage the `Content-Type` of an upload request body instead of inferring the type of an uploaded file based on its content. The M3 operation is effective in bypassing such filtering logic. Because this operation only alters the `Content-Type` value, M3 is applicable to all seed files.

M4: Changing a file extension. This operation changes the `Extension` of a given upload request to one of the seven PHP-style extensions or one of the 17 predefined common extensions. Because FUSE tries every one of these extensions, it produces 19 mutated requests for a given upload request. We observed that web applications often use an extension blacklist to prevent adversaries from uploading malicious files. For this operation, our objective is to try a diverse set of common extensions, including PHP-style extensions, that may invoke a target PHP interpreter or contribute to bypassing content-filtering checks. We collected PHP-style extensions from eight different versions of Apache `mod_php` modules, each of which specifies which extension set invokes a PHP interpreter. For instance, `phar` is a new extension supported by `mod_php` for PHP 7.2. Therefore, developers should update their content-filtering logic as well to block `phar` files. M4 is designed to identify the omission of content-filtering logic for each listed extension. M4 is applicable to all seed types.

M5: Replacing PHP tags. M5 replaces the default PHP opening tags (i.e., `<?php`) in the `Content` of a given upload request with short tags (i.e., `<?`). It is designed to bypass the content-filtering logic that searches only for the default PHP opening tag. M5 is designed solely for a PHP seed file.

M6: Converting HTML into EML. This operation converts the HTML file in the `Content` of a given upload request into

Electronic Mail (EML) [6, 38] by mutating the upload request. EML is the standard format of email files used by Microsoft Outlook, Mozilla Thunderbird, and Apple Mail. Interestingly, an EML file is able to include an HTML document with script elements. M6 first prepends the header of a prepared EML file to the beginning of the `Content`. It then converts HTML special characters in the `Content` to hexadecimal format so that the converted HTML code performs code execution. Finally, it changes the `Extension` of a given seed request into the `eml` extension. We observed that Internet Explorer 9, 10, and 11 execute an HTML in the EML format. Thus, M6 is applicable to (X)HTML seed files.

M7: Removing a file extension. The M7 operation is designed to remove the `Extension` of a given upload request that potentially contributes to bypassing content-filtering checks. Unfortunately, we observed that several web applications do not check the existence of a file extension itself. Since this operation only concerns the `Extension` of the requests, it is suitable for all seed types.

M8: Converting a file to SVG. SVG is a file format in XML that represents a vector image; it facilitates the embedding of HTML code in its file. M8 embeds an HTML file into a prepared Scalable Vector Graphics (SVG) [15] file. M8 appends the start and end tags of a prepared SVG file to the beginning and ending of the `Content` of a given upload request. Additionally, this operation changes the `Extension` of the request to `svg`. Since the SVG file format only supports embedding in an HTML document, the M8 operation is only applicable to HTML files.

M9: Prepending an HTML comment. In this operation, the 4,096 bytes of an HTML comment consisting of an arbitrary string are prepended to the `Content` of a given request. We designed the M9 based on the fact that the content-filtering logic of the XE application checks for the existence of keywords indicating JS scripts or HTML documents in the heading part of an uploaded file. For example, XE searched `<html>` or `<script>` in the heading part but not in the entire file. By leveraging this information, M9 prepends the HTML comment tags (i.e., `<!--, -->`) to the contents of the original HTML seed file, thus bypassing the content-filtering logic. At the same time, because the comment start tags exist in its first 512 bytes, the Chrome and Firefox browsers infer the mutated file to be an executable HTML file. This operation aims to execute an HTML-type file, thus making the operation applicable to both HTML and XHTML seed files.

M10: Changing a file extension to an uncommon extension. M10 changes the `Extension` of a given request to an uncommon extension (e.g., `fuse`). Similar to M4 and M7, this operation is designed to bypass blacklist-based extension filtering checks. We observed that the filtering logic of several web applications does not perform the content-filtering logic for uncommon extensions because they do not know what to check for such uncommon file types. We note that CE of files mutated by M10 depends on whether a web server performs content-sniffing [2]. In the default Apache setting, Apache does not perform content-sniffing for files with uncommon extensions. This invites a browser to infer the file type based on its content by performing content-sniffing. When the browser determines its MIME type to be HTML, the uploaded file becomes executable. This is a classic MIME confusion attack [12]. M10 is suitable for HTML and JS seed files.

M11: Converting a file extension to uppercase. M11 performs an operation that changes the second character in the `Extension` of a given request to uppercase. This operation exploits the discrepancies in checking file extensions between the file filtering logic of a target web application and the type inference module of an Apache server. Consider a target application that allows the upload of an HTML file with the `hTml` extension due to buggy content-filter logic. Now, when a victim accesses this uploaded file, the Apache web server inspects its file extension in the case-insensitive manner and specifies the `Content-Type` header to be the inferred type of `text/html`. The victim's browser executes this file as HTML because of its content header. That is, the target application thinks this file is not an HTML file, but its web server automatically injects the inferred `text/html` MIME type, resulting in CE. M11 is applicable to all seeds.

M12: Prepending a file extension. This operation prepends a given extension to the predefined 14 extensions, including `png`, `jpg`, and `zip`, to the `Extension` of a given seed request. For example, M12 mutates the extension of the uploaded file from `.php` to `.gzip.php` by prepending the `gzip` to the `Extension`. Many applications assess the MIME type of an uploaded file based on its extension to filter out suspicious file types. We designed M12 to deceive the flawed content-filtering logic that infers the MIME type of the file by checking the extension (`gzip`) prepended to the `Extension`, not the original `Extension` (`php`). M12 is applicable to all seeds.

M13: Appending a resource header. M13 appends the eight bytes header of a predefined JPG file to the end of the `Content` of a given upload request. As a result, the uploaded file has two file signatures: one from the original seed file and the other from the predefined JPG file. The goal is to mutate an upload request so that the uploaded file causes a target application to fail to infer the correct MIME type of the file. We observed that the `finfo_file` built-in function returns the two MIME types for this malformed file. M13 abuses this misinterpretation by creating a file with more than one file signature. This operation is applicable to all file types.

VII. Evaluation

We evaluated FUSE for finding U(E)FU vulnerabilities (§VII-B) and compared it against state-of-the-art penetration testing tools (§VII-C). We also analyzed the efficacy of the exercised mutation operations (§VII-D). Finally, we present case studies of the discovered vulnerabilities (§VII-E).

A. *Experimental Setup*

We ran a series of experiments on 33 PHP web applications listed in the first column of Table II. We selected our benchmark applications that support the upload functionality from the three sources: (1) the evaluation set covered by NAVEX [26]; (2) popular CMS applications listed by W3Techs [20]; and (3) highly rated CMS projects in PHP with more than 500 stars on GitHub [8] that report no errors in their installations. According to the W3Techs statistics [20], these are applications with the upload functionality used by at least 5,600 sites [20] or have received large attention from GitHub

Application (Version)	Total # of Attempted Requests	CE			PCE		.htaccess Uploaded	Monitor Enabled	Execution Time
		PHP	HTML	XHTML	PHP	JS			
Bludit(3.8.1)	117,267	0	1	0	3	0	✗	✓	37m 34s
Textpattern (4.7.3)	11	1	1	1	0	1	✗	✗	0s
Joomla (3.9.3)	121,117	0	0	0	28	2	✗	✓	47m 20s
Drupal (8.6.9)	120,849	0	0	0	18	0	✗	✗	70m 39s
CMSMadeSimple (2.2.9.1)	24,986	2	1	1	14	1	✗	✗	22m 53s
Pagekit (1.0.16)	107,609	0	2	1	5	2	✗	✗	36m 59s
Backdrop (1.12.1)	26,930	0	0	0	34	1	✗	✗	17m 16s
CMSimple (4.7.7)	102,168	0	1	0	5	3	✗	✗	19m 3s
WordPress (5.0.3)	98,730	0	4	4	43	8	✗	✗	15m 26s
Concrete5 (8.4.4)	96,638	0	3	2	6	4	✗	✗	38m 59s
Composr (10.0.22)	60	0	1	1	50	1	✗	✓	1s
OctoberCMS‡ (1.0.446)	94,294	0	1	0	5	1	✗	✓	14m 39s
phpBB3 (3.2.5)	119,796	0	0	0	†21 (21)	0	✗	✓	7m 42s
Elgg (2.3.10)	11	1	1	1	0	1	✗	✓	0s
Microweber (1.1.2.1)	47,419	26	39	17	156	13	✗	✗	25m 44s
XE (1.11.2)	105,757	0	†2 (1)	†2 (1)	1	1	✗	✗	325m 51s
SilverStripe (4.3.0)	87,312	0	2	2	8	5	✗	✗	100m 22s
ZenCart (1.5.6a)	121,827	0	1	1	1	1	✗	✓	24m 34s
ECCube3 (3.0.17)	5	1	1	1	0	1	✓	✗	1s
GetSimpleCMS (3.3.15)	52,564	0	9	1	15	12	✗	✗	16m 26s
DotPlant2 (N/A)	5	1	1	1	0	1	✓	✗	1s
MyBB (1.8.19)	12,142	0	†1 (1)	0	†33 (33)	†4 (4)	✗	✓	2m 58s
HotCRP¶ (2.102)	94,034	0	0	0	†3 (3)	0	✗	✗	257m 18s
Subrion (4.2.1)	60	1	1	1	48	1	✗	✗	4s
SymphonyCMS (2.7.7)	24,980	1	1	1	14	1	✓	✗	4m 18s
AnchorCMS (0.12.7)	108,292	0	0	0	4	1	✗	✗	3m 28s
WeBid (1.2.2)	85,317	0	0	0	6	0	✗	✗	19m 42s
Collabtive (3.1)	102,097	0	0	0	1	1	✗	✗	184m 20s
OsCommerce2 (2.3.4.1)	6,825	1	11	1	49	1	✓	✗	10m 31s
X2engine (6.9)	71,021	0	0	0	14	0	✗	✓	71m 38s
ClipperCMS (1.3.3)	63,259	0	1	1	7	1	✓	✗	18m 41s
Monstra (3.0.4)	16,982	2	12	1	15	14	✗	✗	13m 56s
Codiad (2.8.4)	5	1	1	1	0	1	✓	✗	0s

† Includes false positives. False positive numbers are specified in parentheses.

‡ Tested in the PHP 7.0 environment.
¶ Tested in the PHP 7.1 environment.

TABLE II: Evaluation of FUSE.

developers. We intentionally excluded applications with no upload support. Each PHP application differs in its implementation of content-filtering logic. This trend helps test the broad applicability of FUSE in finding U(E)FU vulnerabilities.

Environment. We ran FUSE on a Linux workstation with an Intel core i7-7700 (3.60 GHz) CPU with 32 GB of RAM. For the target system with our benchmarks, we used a Linux workstation with an Intel core i7-8700 (3.20 GHz) CPU with 32 GB of RAM. We installed Ubuntu 16.04, Apache 2.4, and PHP 5.6 at the target system under testing. For some applications that require PHP versions above 5.6, we used a separate Docker container with PHP 7.0 and 7.1. For each PHP interpreter, we deliberately enabled PHP short tags because those short tags are supported by default in PHP versions below 5.3, accounting for 15.1% of web server settings among the Alexa top 10 million websites using PHP [22].

B. Discovering UFU and UEFU Vulnerabilities

Table II summarizes the bugs that FUSE found. The second column describes the total number of upload requests that FUSE attempted. When a chain contributes to triggering UFU or UEFU vulnerabilities, FUSE purges other chains that include this successful chain (§V-B). This mechanism prunes unnecessary upload requests triggering the same vulnerability that a shorter mutation chain has already invoked. *Thus, the number of total requests varies with the number of chains causing successful uploads.* Note that the total number of attempted requests for ECCube3, DotPlant2, and Codiad is five since they allow the upload of the four seeds and the `.htaccess` file, which indicates the absence of content-filtering checks.

The CE column in Table II presents the number of requests that succeeded in finding UEFU vulnerabilities by uploading variants of PHP, HTML, and XHTML. Any positive number in those columns indicates that the corresponding application has UEFU vulnerabilities. For instance, in the case of Microweber, FUSE generated 26 distinctive upload requests, each of which was able to drop an executable PHP file at a target web server. Furthermore, the upload attacker is able to invoke these PHP files with URLs, which enables remote CE.

The PCE column in Table II represents the number of upload requests that succeed in uploading potentially executable PHP and JS files. The eighth column indicates whether an application allows the uploading of a `.htaccess` file. If an application allows a `.htaccess` to upload, we mark it with a ✓, and ✗ otherwise. The ninth column shows whether an application requires the File Monitor. If an application uses the File Monitor, we mark it with a ✓, and ✗ otherwise. 24

applications did not require the presence of the File Monitor. To investigate the feasibility of not applying the File Monitor, we implemented a configuration file to specify the file upload oracle for each application. If placing the File Monitor at the target server for testing is viable, the configuration task becomes much easier, thus rendering FUSE as a gray-box testing tool. The last column shows the execution time for FUSE to finish a penetration testing campaign.

UEFU vulnerabilities. *FUSE reported 30 exploitable UEFU vulnerabilities in 23 applications with 176 distinct upload request payloads*. The 23 vulnerable applications include popular PHP applications, such as WordPress, Concrete5, OsCommerce2, and ZenCart. The estimated number of websites deploying these five applications ranges from 5,600 to three million sites [20].

Instead of reporting each of the 176 distinct requests as one vulnerability, we conservatively counted distinct causes of UEFU vulnerabilities. We leveraged five key objectives (§VI) of mutation operations because each objective aims to exploit a different vulnerability cause. For a list of chains, each of which contributes to producing one successful upload request among the 176 requests, we counted multiple chains with the same mutation objective as one vulnerability. That is, we counted groups of chains with distinct mutation objectives. For example, consider the case that FUSE reports four mutation chains, each of which corresponds to a successful upload request:

$$\{\underbrace{\texttt{M1, M2}}_{\#2}, \underbrace{\texttt{M3}}_{\#5}, \underbrace{\texttt{M4M9}}_{\#2+\#3}\}$$

We count them as three vulnerabilities because the M1 and M2 operations share the same root cause (objective #2) although their upload requests and applied mutations completely differ. The M4M9 chain is a result of two mutation operations with a root cause that is due to developers committing two mistakes (objectives #2 and #3) together. This methodology helps avoid overcounting vulnerabilities that share the same root cause.

We reported all the 30 UEFU vulnerabilities to the corresponding vendors and obtained 15 CVEs from nine applications. Eight vulnerabilities from five vendors have been patched. Five vulnerabilities from four vendors, including WordPress, confirmed that they would address the reported vulnerabilities. 15 bugs are awaiting confirmation from the corresponding vendors. Two vendors declined to patch the reported bugs.

Among the 30 UEFU vulnerabilities, 14 bugs required an administrator-level privilege for their exploitation. We emphasize that for nine of these 14 UEFU vulnerabilities, the implemented content-filtering checks forbid the upload of our seed files for application administrators. Therefore, these UEFU vulnerabilities exhibit mistakes of developers, causing unintended remote CE. Note that a mature web application often limits the upload capability even for their application administrators, thus enforcing the upload of admissible files only, because web host and web application administrators can be different. For instance, a web hosting administrator often separates application administrators from the host management, such as uploading files via SFTP or SCP, and only provides them with access to specified hosting apps [33]. It is known that malicious application administrators have exploited UEFU vulnerabilities to upload web shells to gain access to the host resources [33].

We double-checked whether every uploaded file caused remote CE. Of the 176 upload request payloads, one upload request targeting MyBB and two upload requests for XE were false positives (1.7%). In the case of one MyBB false positive, MyBB appends random tokens in the URL leading to the uploaded file. FUSE is able to retrieve this URL with the help of the File Monitor. In the benchmarks, other applications use randomized URLs and provide these URLs in a web page that an upload attacker can reference and exploit. However, MyBB provides no such page of leaking this randomized URL, thus leaving only one option for the attacker: to guess the URL. Thus, we labeled it as a false positive. The reported URL indeed invokes U(E)FU vulnerabilities; however, this does not account for the fact that the URL is difficult for the attacker to guess. The two false positives for XE involved uploading an HTML and an XHTML file after applying M6. We found that XE removes the extension (`.eml`) of an uploaded file, thereby rendering the web server unable to infer the MIME type when setting the `Content-Type` header to the response. This enforces a browser fetching this resource to infer the fetched resource type via content-sniffing. We tested these uploaded files with Chrome, Firefox and Internet Explorer. Every browser rendered them as text files, thus resulting in no execution. Conducting this additional verification of running Internet Explorer on such uploaded files can eliminate these two false positives. However, introducing this additional step makes our tool to depend on various headless browser execution environments. Validating `Content-Type` headers (§V-D) without this step meets our goal with few false positives.

UFU vulnerabilities. *FUSE found 55 UFU vulnerabilities from 30 applications with 630 distinct upload request payloads.* Among the 630 requests, which excluded 176 requests that trigger UEFU vulnerabilities from the total of 806 successful upload requests, we counted UFU vulnerabilities by applying the same counting standard outlined above. Because we excluded upload requests that trigger UEFU vulnerabilities, each one of the 55 UFU vulnerabilities cannot become a UEFU vulnerability. Table II shows that 30 applications (91%) in our benchmarks have at least one UFU vulnerability posing a risk of PCE. This demonstrates that their emplaced content-filtering logic is unable to prevent an attacker from uploading executable PHP and JS files.

We verified whether all of the uploaded PHP and JS files were indeed executable. We placed our own webpage with an LFI vulnerability at the web server and conducted LFI attacks against it to execute each uploaded PHP file. For each JS file, we made another webpage including the script tag with a source URL that leads to the uploaded JS file. We then visited this page to check the execution of the JS files with Chrome, Firefox, and Internet Explorer browsers.

Of the 630 upload requests in the PCE column, 61 upload requests targeting the HotCRP (0.4%), phpBB3 (3.0%), and MyBB (5.4%) applications were false positives (8.8%). For HotCRP, three reported requests were false positives. Since HotCRP stores the uploaded file in its database instead of using a file system, we could not perform the LFI attack.

Vulnerability (Risk)	FUSE	fuxploider	UploadScanner
UEFU (PHP CE)	12	7	5
UEFU (HTML CE)	23	N/A	14
UFU (JS PCE)	26	N/A	21

N/A: not applicable for HTML and JS files

TABLE III: The number of unique U(E)FU vulnerabilities found from the benchmarks using three different testing tools.

This means that it is feasible to upload an executable PHP file; however, we do not have a sink method to trigger its execution. Both phpBB3 and MyBB use random tokens in the URLs of uploaded files, which renders an attacker unable to guess these URLs.

We observed that ECCube3, DotPlant2, SymphonyCMS, OsCommerce2, ClipperCMS, and Codiad allow the upload of a `.htaccess` file, which entails a security-critical consequence. Now, the adversary is capable of inducing an Apache web server to invoke a PHP interpreter for any file extension, which allows the PHP interpreter to execute any uploaded file. For instance, the adversary is able to upload 49 unique PHP variants for OsCommerce2, as Table II shows. These uploaded files impose the risk of PCE. However, the adversary can reprogram a `.htaccess` file, and make a PHP interpreter to be invoked for each of the 49 PHP variants, which enables CE. Thus, we reported all findings regarding `.htaccess` uploading bugs to the vendors and obtained two CVEs from the OsCommerce2 and ClipperCMS.

Performance. The execution times of FUSE vary with the target applications because FUSE invokes application-specific upload functionality. Elgg, ECCube3, DotPlant2, and Codiad took less than two seconds because they allowed the upload of all the four seed files. For such cases, FUSE does not attempt to find more complicated examples because they implement no content-filtering checks (§V-B). On the other hand, FUSE took more than 100 minutes to complete a penetration testing campaign on XE, SilverStripe, HotCRP, and Collabtive. These delays emanated from their internal implementation of handling concurrent sessions associated with requests. They used the PHP session built-in methods that often hang upon locking a session file until on-going requests unlock the session file [13]. That is, these applications are not designed to handle bulk requests from one session. Other applications implement their own session handling methods or explicitly unlock the session file before generating a response completes.

C. Comparison against State-of-the-Art Penetration Testing Tools

We compared FUSE against two state-of-the-art tools: fuxploider [7], and UploadScanner [19]. Fuxploider is an open-source upload vulnerability scanning tool and UploadScanner [19] is an extension for Burp Suite Pro, a commercial platform for web application security testing. We selected these tools because they are penetration testing tools available from GitHub and are specifically designed to find U(E)FU vulnerabilities.

We ran both the scanners on the same benchmarks and counted vulnerabilities by applying the same counting standard aforementioned. We manually examined each successful exploit and its cause from the reported bugs. Table III summarizes the vulnerabilities found by each tool. Note that while fuxploider only attempts the upload of PHP files, UploadScanner uploads PHP and HTML files to trigger CE as well as JS files to trigger PCE. For fair comparison, we compared the performance of FUSE for each seed file type.

PHP CE. With regard to uploading PHP files, FUSE found more than twice as many vulnerabilities as fuxploider and UploadScanner found. Fuxploider missed five UEFU vulnerabilities from five applications due to several implementation issues. For instance, the tool generates an execution error when a target application generates an upload response with the `content-encoding` header to be `gzip`. Fuxploider is also unable to retrieve randomized URLs for checking the presence of uploaded files, which the File Monitor of FUSE is able to support.

FUSE also found seven more UEFU bugs than UploadScanner. Four UEFU bugs stem from the capability of FUSE considering diverse PHP-style extensions, including `pht` and `php7`, when applying M4. However, UploadScanner only tries two extensions for penetration testing: `php5` and `phtml`, thus failing to find those four bugs. The remaining three UEFU bugs are due to the incapability of retrieving randomized URLs and case-sensitive comparison for matching file names. UploadScanner only computes an upload URL from a predefined file name. When a target application changes this file name to lowercase, UploadScanner becomes unable to check the successful upload of this file, thus producing a false negative.

HTML CE. For UEFU bugs involving HTML files, FUSE found nine more bugs than UploadScanner. Seven of the nine bugs were due to the File Monitor module and the aforementioned miscellaneous implementation issues of UploadScanner. The remaining two bugs were found by the M9 and M13 operations. For example, FUSE found a UEFU bug from WordPress by trying the combination of M4 and M13, while UploadScanner was unable to identify the bug.

JS PCE. Regarding UFU bugs with the JS seed, there are 19 common bugs between FUSE and UploadScanner. In particular, UploadScanner missed seven bugs due to the same aforementioned issues and the inability of leveraging the File Monitor. From two applications, FUSE missed two bugs that involves injecting JS payloads into the GIF metadata because FUSE did not apply M2 to JS files. These false negatives are easily fixable by revising the conflict rules among the mutation operations.

D. Effectiveness of Mutations

Operation significance. Figure 7 illustrates the frequency of each mutation. Each histogram corresponds to a mutation operation, and its height represents the total number of successful upload requests that have used this mutation. We observed that every mutation was used to generate at least five upload requests that triggered UFU vulnerabilities. This demonstrates that every mutation is indispensable. Note that the M4 operation significantly outperformed other operations by achieving the highest frequency. Recall that applying the M4 operation means producing an upload request for each extension, resulting in 19 different requests, each attempting to forge its own extensions. The effectiveness of the M4

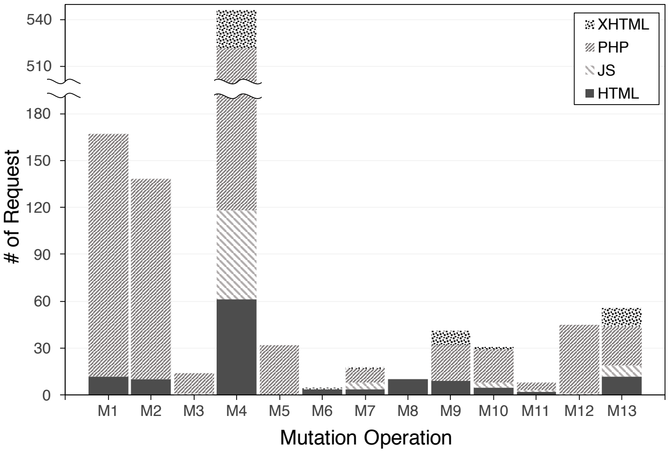


Fig. 7: The frequency of successful mutation operations in triggering U(E)FU vulnerabilities.

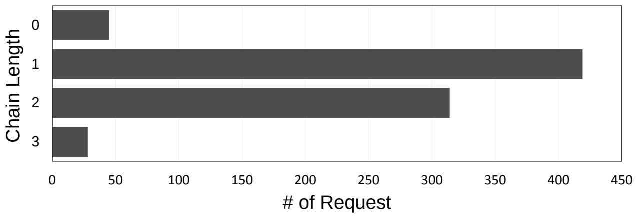


Fig. 8: The chain length frequency of successful chains.

operation also indicates that many applications implement buggy content-filtering checks based on file extensions.

Chain length. We also measured the frequency of each chain length that triggered U(E)FU vulnerabilities. As Figure 8 shows, FUSE reported 45, 419, 314, and 28 upload requests with chain lengths of zero, one, two, and three, respectively.

A chain length of zero indicates that an upload request with no mutation triggers a UFU vulnerability. We observed that FUSE found many bugs by applying a single mutation, which demonstrates that each mutation is quite effective at bypassing content-filtering checks. We also observed that 28 upload requests that triggered UFU vulnerabilities resulted from applying a chain with a length of three. Considering that FUSE coordinates the shortest chains to trigger UFU vulnerabilities, the existence of those long chains implies the difficulty of manually finding these bugs.

Vulnerability causes. Table IV presents the number of vulnerabilities that FUSE found after applying the mutations with their respective mutation objectives. The column for the first objective shows that FUSE found 14 UFU and 13 UEFU vulnerabilities, including four CVEs resulting from the absence of any content-filtering checks. The mutations, designed for the third objective, are the most effective, contributing to finding 21 UFU and 14 UEFU vulnerabilities, including 11 CVEs. FUSE reports these objectives along with actual payload exploits that help users understand the root causes of the discovered U(E)FU vulnerabilities.

Constraint consistency. Note that the 13 mutation operations are designed to preserve the execution semantic of the seed files. However, when target execution environments, including web browsers and PHP interpreters, change their execution constraints due to their software updates, these mutation operations should reflect such changes.

	#1	#2	#3	#4	#5	#2+#3	#2+#3+#4
UEFU (CVE)	13 (4)	0	14 (11)	1	0	2	0
UFU$\setminus$UEFU	14	5	21	5	5	4	1
Total	27	5	35	6	5	6	1

TABLE IV: Causes of the identified U(E)FU vulnerabilities.

We tested how execution constraints remain consistent across different versions of browsers and PHP interpreters. Specifically, we checked whether an executable file mutated from one seed file remains executable across different execution environments. For the PHP, HTML, XHTML, and JS seed files, we applied all combination chains of the 13 mutations of which the length is less than three, thus preparing a set of mutated seed files. We then tested whether they were executable across different versions of browsers and PHP interpreters. For this experiment, we deployed Chrome (versions 53, 61, 69, and 77), Firefox (versions 49, 52, 62, and 69), Internet Explorer (versions 9, 10, and 11) and Safari (versions 10, 11, 12, and 13) for checking the execution constraints of JS and (X)HTML variants. For each browser, we picked the stable version released at every October in the last four years (2016-2019). We also tested PHP variants against different versions of PHP interpreters and Apache `mod_php` modules enabling the PHP short tags (versions 5.2, 5.6, 7.0, 7.1, 7.2, and 7.3).

We observed that *all executable JS and (X)HTML variants remain consistent across different versions of the browsers except for one case.* It denotes that the extracted constraints do not change much across the different versions of these browsers, requiring no change in our mutations as these software evolve. The one anomalous case arose from Internet Explorer 9. A JS file mutated by M2 is executable by Internet Explorer 10 and 11. However, Internet Explorer 9 treats the JS payload in the metadata section of this image/JS file as an unterminated comment, resulting in no execution. However, this JS file is executable by every version of Chrome, Firefox, and Safari, indicating that the JS file will be executable when a victim uses Chrome, Firefox, or Safari.

Note that all mutated (X)HTML and JS files remain consistent across the four different versions of Safari, which we did not analyze when designing the 13 mutations. This observation demonstrates that the extracted constraints are browser-agnostic in that no change is required to generate upload requests that would drop Safari-executable (X)HTML and JS files.

The execution of PHP files with different PHP-style extensions varies across PHP versions. For instance, the direction invocation and execution of a PHP file with `phar` via URL is only feasible in PHP 7.2 and 7.3. Besides the differences stemming from extensions, the execution results of the mutated PHP files causing PCE are consistent across the different PHP interpreters.

We concluded that browser updates have little impact on the capability of FUSE generating executable upload files. As for PHP interpreter updates, FUSE may need to cover more

PHP-style extensions with interpreter updates over time.

E. Case Studies

In the following, we investigate the findings of FUSE and how its mutation operations contributed towards uncovering these bugs. We specifically focus on the UEFU bugs from Concrete5, Joomla, and Microweber.

Concrete5. Figure 9 shows an uploaded SVG file that invokes CVE-2018-19146 in Concrete5. This uploaded file is a result of the mutated upload request, which is the result of applying the M8 operation to the HTML seed request. Concrete5 allows users to upload images and considers an SVG file as an image, as Figure 10 shows. However, the whitelist allows the adversary to upload SVG files with the HTML code embedded with an arbitrary JS script, which causes CE.

```
<svg>
  <html>
    <head><title>test</title></head>
    <body><script>alert('xss');</script></body>
  </html>
</svg>
```

Fig. 9: A simplified SVG file with injected HTML code (M8.svg).

```
'upload' => [
 'extensions' =>
  '*.ppt;*.pptx;*.kml;*.xml;*.svg;*.webm;'.
  ...
]
```

Fig. 10: The whitelist of acceptable upload file extensions in Concrete5.

Joomla. Joomla implements strict content-filtering logic that does not permit the upload of any file with PHP scripts, thus preventing PCE. FUSE generated an uploading request that successfully dropped an executable PHP, as shown in Figure 11. This uploaded file is the result of applying the M1, M4, and M5 operations together. Leaving out any of these mutation operations results in not bypassing content-filtering checks in Joomla. This case demonstrates that FUSE is capable of generating a complicated input that triggers erroneous behaviors in a target web application. It also shows that applying a single mutation is not enough to find a UFU vulnerability.

```
\x89\x0d\x0a\x1a\x0a\x00\x00\x00\x0d
\x49\x48...  #1024 bytes of PNG binary
<?
  $sn = pack('H*', dechex(2534024256545858215*2));
  echo $sn;
  # $sn set to "FUSE_GEN" after the Ln #4.
  ...
?>
```

Fig. 11: A simplified PHP exploit posing a PCE risk (M1PNG_M4GIF_M5.gif) against Joomla.

Microweber. Figure 12 shows the variant of a seed PHP file after applying the M4 and M13 operations together. Microweber internally manages a blacklist of file extensions and MIME types. Thus, from each upload request, Microweber extracts a file extension and infers the MIME type, and then matches it to the blacklist. This case abuses the pht extension, which is not on the blacklist, and changes the inferred MIME type to be application/octet-stream by appending the JPG header to its content. Both M4 and M13 are essential to trigger this UEFU vulnerability, which causes CE.

```
<?php
  $sn = pack('H*', dechex(2534024256545858215*2));
  echo $sn;
?>
\xff\xd8\xff\xee\x00\x10JF    # JPG file signature
```

Fig. 12: A simplified PHP exploit (M4PHT_M13.pht) against Microweber.

VIII. Limitation and Discussion

We presented the five objectives that capture common developer mistakes and implemented 13 mutations. We acknowledge that there exist other mutation methods that achieve the same objectives. However, the presented objectives are general enough for users to suggest their own mutations that achieves the same goal. For instance, the second mutation objective is to cause incorrect type inferences by manipulating Content. We triggered incorrect type inferences for the finfo_file built-in function. However, a user can implement a different mutation that would bring the same result and integrate it with FUSE to decrease possible false negatives.

We manually examined the execution constraints of the browsers and PHP interpreters (§VI) and reflected those constraints when designing the 13 mutations. Therefore, when these constraints embedded in these software change due to their updates, the mutations should also be modified to reflect these changes (§VII-D). The automatic extraction of these execution constraints [30] and the reflection of such constraints on mutations are interesting technical challenges that we leave to future research.

We observed that the most common mistake causing UFU vulnerabilities was using an incomplete blacklist or flawed whitelist of extensions. This trend stems from the ignorance of developers with respect to file types posing a low security risk. For example, it requires domain-specific expertise to know that SVG and EML files are able to execute embedded scripts. Furthermore, file extensions embedded in upload requests are usually under the control of upload attackers. Thus, inferring upload file types based on user-provided extensions opens a door for further attacks. Developers should check the actual content of a given file to determine its admissibility [18].

Another vulnerability source was due to smart browsers performing content-sniffing. Assume that an attacker attempts the upload of an attack file that a target application accepts. In some cases, the Apache server hosting the application is unable to infer the uploaded file type, thus placing no Content-Type header in the response to a request for this file. This invites a browser to infer the file type based on its content by performing content-sniffing, which the upload attacker exploits. For uploaded files of which a web server cannot infer their types, we recommend setting the X-Content-Type-Options header to enable nosniff, which prevents browsers from performing content-sniffing [11]. Adjusting an Apache configuration file to setup the default value for this header blocks the attack. Also, web applications can specify the header with the specific file type that the application inferred, thus preventing the attack.

IX. Related Work

MIME confusion attack. Barth *et al.* [30] proposed content-sniffing XSS attacks, which targets discrepancies between a web browser and the content file filtering logic of a target website. They demonstrated that a stored XSS is possible by exploiting uploaded PDF files. However, they covered a subset of UEFU vulnerabilities that exploit the content-sniffing algorithms of major browsers. FUSE considered more diverse attack vectors that enable CE via file uploads, such as placing attack code in SVG files and uploading images that contain attack PHP code. Jana *et al.* [41] presented chameleon attacks that exploit discrepancies in file type inference heuristics between the malware detector in a remote environment and file parsing algorithms in the actual host application.

Moreover, there have been numerous attempts to find content-sniffing XSS attacks by leveraging PNG or PDF chameleons [9, 24, 52]. Our framework is inspired by the approaches of these works, but our goal is to evade the file filtering logic in CMS web applications that rarely parses files. Also, we considered many attack vectors in our mutation operations that can trigger U(E)FU vulnerabilities in addition to the chameleon attack.

Finding web vulnerabilities. Previous research proposed static analyses in identifying data-flow vulnerabilities, including XSS and SQL injection [43, 49, 63, 65]. Backes *et al.* [28] presented a scalable framework for computing code property graphs [66] from PHP web applications. The authors leveraged graph traversal on the computed graphs to identify XSS and SQLI vulnerabilities. Doupé *et al.* [35] and Payet *et al.* [55] identified EAR vulnerabilities, which are control-flow bugs that allow continuous execution after redirection. Lee *et al.* [46] manually analyzed progressive web applications in terms of their security and privacy and reported new ways of abusing unique progressive web features.

Saner [29] validates the safety of custom sanitization routines. It statically approximates string values that a variable can hold at certain program points with an automata instance and then checks the feasibility of accepting escaping characters. For the subsequent step, it then dynamically injects attack strings in a pre-defined test suite to remove false positives. This approach is clearly applicable to finding U(E)FU vulnerabilities. However, their method requires modeling diverse PHP built-in functions as transducers, which requires non-trivial engineering efforts.

There are several works on applying symbolic execution to PHP web applications [26, 40, 59, 62, 65]. Huang *et al.* [40] conducted symbolic execution to discover UEFU vulnerabilities allowing the upload of PHP files. They modeled PHP built-in functions regarding file writing functionality (i.e., `move_uploaded_file` or `file_put_content`) as a vulnerable sink, and they devised a reachability constraint to guarantee that such functions are reachable from a tainted source (i.e., `$_FILES`). They also designed an extension constraint to ensure that the uploaded PHP file indeed has the PHP-style file extensions. Thus, they aimed to check whether an arbitrary file can be uploaded with the PHP-style file extensions. They evaluated their tool on 9,160 WordPress plugins and found only three vulnerabilities. On the other hand, FUSE takes into account multiple mutation vectors other than the `Extension`, such as `Content-Type` and `Content`, which should be considered to find sophisticated U(E)FU vulnerabilities from 33 applications.

NAVEX [26] introduced an automatic exploit generation framework. It combines static and dynamic analyses to identify the paths from sources to vulnerable sinks while considering sanitization filters and generates exploit strings by solving symbolic constraints. Son and Shmatikov [59] presented SAFERPHP for discovering semantic bugs by leveraging taint analysis and symbolic execution. THAPS [42] is a web scanner that applies symbolic execution to simulate all possible execution paths and carry out a taint analysis as a post-process for finding defects. Sun *et al.* [62] conducted symbolic execution with taint analysis to identify logical vulnerabilities in e-commerce applications. Their tool explores critical logic states, which include payment status, across checkout nodes.

X. Conclusion

We propose FUSE, a penetration testing tool designed to find U(E)FU vulnerabilities. We present 13 mutation operations that transform executable seed files to bypass content-filtering checks while remaining executable in target execution environments. We evaluated FUSE on 33 real-world PHP applications. FUSE found 30 UEFU vulnerabilities including 15 CVEs, which demonstrates the practical utility of FUSE in finding code execution bugs via file uploads.

Acknowledgments

The authors would like to thank the anonymous reviewers for their concrete feedback. This work was supported by National Research Foundation of Korea (NRF) Grant No.: 2017073934.

References

[1] "Apache HTTP server tutorial: .htaccess," https://httpd.apache.org/docs/2.2/en/howto/htaccess.html.

[2] "Apache module mod_mime_magic," http://httpd.apache.org/docs/2.4/mod/mod_mime_magic.html.

[3] "Arachni web application security scanner framework," http://www.arachni-scanner.com/.

[4] "Broken access control," https://www.owasp.org/index.php/Broken_Access_Control.

[5] "Burp suite - cybersecurity software from portswigger," https://portswigger.net/burp.

[6] "Email (electronic mail format)," https://www.loc.gov/preservation/digital/formats/fdd/fdd000388.shtml.

[7] "fuxploider," https://github.com/almandin/fuxploider.

[8] "Github PHP CMS project," https://github.com/topics/php?o=desc&q=cms&s=stars.

[9] "The hazards of MIME sniffing," https://adblockplus.org/blog/the-hazards-of-mime-sniffing.

[10] "Joomla," https://www.joomla.org/.

[11] "Mdn web docs: X-content-type-options," https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/X-Content-Type-Options.

[12] "Mitigating mime confusion attacks in firefox," https://blog.mozilla.org/security/2016/08/26/mitigating-mime-confusion-attacks-in-firefox/.

[13] "PHP session locking: How to prevent sessions blocking in PHP requests," https://ma.ttias.be/php-session-locking-prevent-sessions-blocking-in-requests/.

[14] "SQLmap: automatic sql injection and database takeover tool," http://sqlmap.org/.

[15] "SVG file format reference," https://www.w3.org/TR/SVG2/intro.html#AboutSVG.

[16] "Testing for local file inclusion," https://www.owasp.org/index.php/Testing_for_Local_File_Inclusion.

[17] "Testing for stored cross site scripting," https://www.owasp.org/index.php/Testing_for_Stored_Cross_site_scripting_(OTG-INPVAL-002).

[18] "Unrestricted file upload," https://www.owasp.org/index.php/Unrestricted_File_Upload.

[19] "Uploadscanner burp extension," https://github.com/PortSwigger/upload-scanner.

[20] "Usage of content management systems for websites," https://w3techs.com/technologies/overview/content_management/all.

[21] "Usage of server-side programming languages for websites," https://w3techs.com/technologies/overview/programming_language/all.

[22] "Usage statistics and market share of PHP for websites," https://w3techs.com/technologies/details/pl-php/all/all.

[23] "Wordpress," https://wordpress.org/.

[24] "XSS-exploit door microsoft betiteld als 'by design'," https://tweakers.net/nieuws/47643/xss-exploit-door-microsoft-betiteld-als-by-design.html.

[25] "ZAP: The OWASP zed attack proxy," https://www.zaproxy.org/.

[26] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan, "NAVEX: precise and scalable exploit generation for dynamic web applications," in *Proceedings of the USENIX Security Symposium*, 2018, pp. 377–392.

[27] D. Babic, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2011, pp. 12–22.

[28] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi, "Efficient and flexible discovery of PHP application vulnerabilities," in *Proceedings of the IEEE European Symposium on Security and Privacy*, 2017, pp. 334–349.

[29] D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Saner: Composing static and dynamic analysis to validate sanitization in web applications," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2008, pp. 387–401.

[30] A. Barth, J. Caballero, and D. Song, "Secure content sniffing for web browsers, or how to stop papers from reviewing themselves," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2009, pp. 360–371.

[31] A. Barth, C. Jackson, and J. C. Mitchell, "Securing frame communication in browsers," in *Proceedings of the USENIX Security Symposium*, 2008, pp. 17–30.

[32] J. Bau, E. Bursztein, D. Gupta, and J. Mitchell, "State of the art: Automated black-box web application vulnerability testing," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2010, pp. 332–345.

[33] D. Canali, D. Balzarotti, and A. Francillon, "The role of web hosting providers in detecting compromised websites," in *Proceedings of the International Conference on World Wide Web*, 2018, pp. 177–188.

[34] M. Dalton, C. Kozyrakis, and N. Zeldovich, "Nemesis: Preventing authentication and access control vulnerabilities in web applications," in *Proceedings of the USENIX Security Symposium*, 2009, pp. 267–282.

[35] A. Doupé, B. Boe, C. Kruegel, and G. Vigna, "Fear the EAR: discovering and mitigating execution after redirect vulnerabilities," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2011, pp. 251–262.

[36] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna, "deDacota: Toward preventing server-side XSS via automatic code and data separation," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2013, pp. 1205–1216.

[37] D. Endler, "The evolution of cross site scripting attacks," iDEFENSE Labs, Tech. Rep., 2002.

[38] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (MIME) part one: Format of internet message bodies," Tech. Rep., 1996.

[39] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated whitebox fuzz testing," in *Proceedings of the Network and Distributed System Security Symposium*, 2008, pp. 151–166.

[40] J. Huang, Y. Li, J. Zhang, and R. Dai, "UChecker: Automatically detecting php-based unrestricted file upload vulnerabilities," in *Proceedings of the International Conference on Dependable Systems Networks*, 2019, pp. 581–592.

[41] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2012, pp. 80–94.

[42] T. Jensen, H. Pedersen, M. C. Olesen, and R. R. Hansen, "THAPS: automated vulnerability scanning of PHP applications," in *Proceedings of the Nordic Conference on Secure IT Systems*, 2012, pp. 31–46.

[43] N. Jovanovic, C. Kruegel, and E. Kirda, "Pixy: a static analysis tool for detecting web application vulnerabilities," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2006, pp. 258–263.

[44] S. Kals, E. Kirda, C. Kruegel, and N. Jovanovic, "SecuBat: a web vulnerability scanner," in *Proceedings of the International Conference on World Wide Web*, 2006, pp. 247–256.

[45] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[46] J. Lee, H. Kim, J. Park, I. Shin, and S. Son, "Pride and prejudice in progressive web apps: Abusing native app-like features in web applications," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2018, pp. 1731–1746.

[47] S. Lekies, K. Kotowicz, S. Groß, E. V. Nava, and M. Johns, "Code-reuse attacks for the web: Breaking cross-site scripting mitigations via script gadgets," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2017, pp. 1709–1723.

[48] Y. Li, B. Chen, M. Chandramohan, S.-W. Lin, Y. Liu, and A. Tiu, "Steelix: program-state based binary fuzzing," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2017, pp. 627–637.

[49] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2009, pp. 75–86.

[50] L. Masinter, "Returning values from forms: multipart/form-data," Tech. Rep., 2015.

[51] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of UNIX utilities," *Communications of the ACM*, vol. 33, no. 12, pp. 32–44, 1990.

[52] G. Molnár and K. Kotowicz, "Content sniffing with comma chameleon," *PoC||GTFO*, vol. 12, no. 4, pp. 14–27, 2016.

[53] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans, "Automatically hardening web applications using precise tainting," in *Proceedings of the Information Security Conference and Privacy*, 2005, pp. 295–307.

[54] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou, "CSPAutoGen: Black-box enforcement of content security policy upon real-world websites," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 653–665.

[55] P. Payet, A. Doupé, C. Kruegel, and G. Vigna, "EARs in the wild: large-scale analysis of execution after redirect vulnerabilities," in *Proceedings of the ACM Symposium on Applied Computing*, 2013, pp. 1792–1799.

[56] G. Pellegrino, O. Catakoglu, D. Balzarotti, and C. Rossow, "Uses and abuses of server-side requests," in *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses*, 2016, pp. 393–414.

[57] J. Ruderman, "Same Origin Policy (SOP)," http://www.mozilla.org/projects/security/components/same-origin.html.

[58] S. Son, K. S. McKinley, and V. Shmatikov, "RoleCast: Finding missing security checks when you do not know what checks are," in *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, 2011, pp. 1069–1084.

[59] S. Son and V. Shmatikov, "SAFERPHP: Finding semantic vulnerabilities in php applications," in *Proceedings of the ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2011.

[60] S. Stamm, B. Sterne, and G. Markham, "Reining in the web with content security policy," in *Proceedings of the International Conference on World Wide Web*, 2010, pp. 921–930.

[61] B. Sterne and A. Barth, "Content Security Policy (CSP)," https://dvcs.w3.org/hg/content-security-policy/raw-file/tip/csp-specification.dev.html.

[62] F. Sun, L. Xu, and Z. Su, "Detecting logic vulnerabilities in e-commerce applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2014.

[63] G. Wassermann and Z. Su, "Sound and precise analysis of web applications for injection vulnerabilities," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, 2007, pp. 32–41.

[64] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc, "CSP is dead, long live CSP! on the insecurity of whitelists and the future of content security policy," in *Proceedings of the ACM Conference on Computer and Communications Security*, 2016, pp. 1376–1387.

[65] Y. Xie and A. Aiken, "Static detection of security vulnerabilities in scripting languages," in *Proceedings of the USENIX Security Symposium*, 2006, pp. 179–192.

[66] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, "Modeling and discovering vulnerabilities with code property graphs," in *Proceedings of the IEEE Symposium on Security and Privacy*, 2014, pp. 590–604.