Remote Code Execution from SSTI in the Sandbox: Automatically Detecting and Exploiting Template Escape Bugs

Yudi Zhao[1,¶], Yuan Zhang[1,¶], and Min Yang[1]

[1]*School of Computer Science, Fudan University, China*
[¶]*co-first authors*

Abstract

Template engines are widely used in web applications to ease the development of user interfaces. The powerful capabilities provided by the template engines can be abused by attackers through server-side template injection (SSTI), enabling severe attacks on the server side, including remote code execution (RCE). Hence, modern template engines have provided a sandbox mode to prevent SSTI attacks from RCE.

In this paper, we study an overlooked sandbox bypass vulnerability in template engines, called template escape, that could elevate SSTI attacks to RCE. By escaping the template rendering process, template escape bugs can be used to inject executable code on the server side. Template escape bugs are subtle to detect and exploit, due to their dependencies on the template syntax and the template rendering logic. Consequently, little knowledge is known about their prevalence and severity in the real world. To this end, we conduct the first in-depth study on template escape bugs and present TEFuzz, an automatic tool to detect and exploit such bugs. By incorporating several new techniques, TEFuzz does not need to learn the template syntax and can generate PoCs and exploits for the discovered bugs. We apply TEFuzz to seven popular PHP template engines. In all, TEFuzz discovers 135 new template escape bugs and synthesizes RCE exploits for 55 bugs. Our study shows that template escape bugs are prevalent and pose severe threats.

1 Introduction

Template is a specialized programming language designed to develop web interfaces [13]. With a simplified syntax, template significantly reduces the bar for web application developers and facilitates the development of server-side applications. By designing the HTML views in templates, web application developers use template engines (TE) to render dynamic HTML views from the templates. With the help of templates and TEs, it is easier to separate the view code from the business code in web application development. Therefore, TEs are widely used in web applications, such as wikis, blogs, and content management systems (CMS) [15].

To support template development, TEs provide plenty of programmable interfaces. Since the template code runs on the server side, templates also open a new attack surface. By injecting payloads into a template, an attacker could abuse the capabilities provided by the TEs, causing severe attacks on the server side, such as remote code execution (RCE). This kind of attack is known as server-side template injection (SSTI), a popular and well-known attack vector [12].

The root cause of SSTI is that an attacker gains the capability of controlling template code. However, template code injection is hard to avoid, due to the nature of TEs, i.e., rendering template code according to external inputs [4, 7]. Moreover, template code modification can be achieved through other vulnerabilities, e.g., file upload vulnerability [6, 8], and is usually provided as standard functionality in many web applications, e.g., style customization. Another underlying cause of SSTI attacks is that TEs provide template code with much more capabilities than really needed, while these powerful capabilities might be abused by an external attacker through template code modification. In light of this, to prevent SSTI attacks, modern template engines (e.g., Smarty and Twig) provide a sandbox mode. In general, the sandbox mitigation mechanism significantly constrains the capabilities acquired from the template code. Hence, under the sandbox mode, SSTI attacks become pretty hard to achieve RCE.

In this paper, we study an overlooked vulnerability in TEs that could bypass the sandbox mode and elevate SSTI attacks to RCE. This vulnerability occurs during the template rendering process. Taking PHP as an example, most PHP TEs adopt a generation-based design in template rendering. These TEs translate a template file into a PHP file and execute the generated PHP file for HTML view rendering. By caching the generated PHP files, the template rendering process achieves near native performance to the PHP code. However, due to the improper implementation of the TEs, the template code may escape the template semantic during the rendering, leading to PHP code injection into the generated PHP files. We name this vulnerability as template escape.

With the capability of PHP code injection, template escape bugs cause severe consequences. However, due to the dependencies on the template syntax and the template rendering logic, template bugs are subtle to detect and exploit. Therefore, little knowledge is known about their prevalence and severity in real-world TEs. To this end, we aim to conduct an in-depth study on template escape bugs. Our study requires an automatic tool to detect template escape bugs and assess their exploitability. However, it is non-trivial to automatically detect and exploit template escape bugs due to several reasons. First, the root cause of the bug lies in the template rendering logic, which heavily intertwines with the syntax parsing and is hard to understand and reason automatically. Second, the input to trigger and exploit a template escape bug is a highly-structured input with grammar. It is challenging to construct such inputs. Third, unlike memory corruption bugs, template escape bugs are semantic errors and hard to identify.

In light of these limitations, we propose TEFUZZ, a tailored fuzzing-based framework to facilitate the detection and exploitation of template escape bugs. The advantages of dynamic testing are two-fold: it not only avoids understanding the complicated template syntax but also takes the generated PHP code to facilitate bug detection and exploitation. The major technical challenges of TEFUZZ are how to guarantee testing coverage and meanwhile reduce redundant testing. Technically, TEFUZZ incorporates several important designs to make the bug detection quite effective, such as balancing the exploration and the exploitation, leveraging PHP syntax to guide PoC generation, leveraging coverage information to cluster similar testcases, and leveraging feedback to adapt failed testcases. In short, by collecting a set of well-structured template files as seeds, TEFUZZ first mutates these seeds to discover potential escape points in them (named as interesting testcases) and then looks for the right payloads that could leverage these interesting testcases to escape the template semantic (i.e., finding the PoCs). By fixing the escaped context of the PoCs in the generated PHP code, TEFUZZ further turns PoCs into RCE exploits.

We apply TEFUZZ to seven popular PHP TEs. TEFUZZ detects 135 template escape bugs in six TEs, showing that template escape bugs are quite prevalent. Further, TEFUZZ successfully exploits 55 bugs, covering every vulnerable TE. Moreover, we have verified the full exploitability of the synthesized RCE exploits with 11 (five N-day and six 0-day) real-world SSTI vulnerabilities, indicating template escape bugs as severe threats. Besides, our evaluation shows that none of these bugs could be detected by existing SSTI scanners. Based on the discovered bugs, we also shed light on the root causes of these less-understood bugs and compare the practices among TEs in rendering templates. At last, we measure the usefulness of each design in TEFUZZ. The results show that all the incorporated designs significantly help TEFUZZ to guarantee testing coverage while reducing testing scope.

In summary, this paper makes the following contributions:

- We study an overlooked and severe sandbox bypass vulnerability in template engines and demonstrate its root cause.
- We present an automatic tool to detect and exploit template escape bugs and introduce several new techniques.
- We discover 135 bugs in seven PHP template engines and construct 55 exploits that enable RCE attacks.

2 Background

2.1 Template Engine

Template Engine (TE) facilitates the development of server-side web applications. With the support of a TE, web applications generate dynamic HTML views from the templates. In general, there are two kinds of content in a template: the HTML content that would be directly output to the front-end and the template code that defines the instructions for generating HTML content. Given a template, TE first parses its syntax and then renders it into HTML content by following the instructions defined in the template code.

Compared with PHP code, template code is more convenient to design front-end views, owing to its simplified language features and optimized programming interfaces. Taking PHP as an example, we find that more than 65% of popular PHP applications on GitHub use TEs to generate their front-end views. Popular TEs include Smarty [1], Twig [2], etc. There are many different TEs, but their syntax is quite similar. To be specific, template code is usually defined within tags. An obvious difference among TEs is the delimiter they used to mark tags. For example, Smarty uses '`{`' and '`{*`' as the delimiters while Twig uses "`{{`", "`{#`" and '`{%`". Further, TEs usually provide a different set of tags. For example, Smarty has a `{config_load}` tag for loading configuration variables from an external file while other TEs do not.

Though there are various tags, they can be divided into three categories: *comment*, *variable*, and *function*. ❶ Comment tags are only used in the templates while not output to the front-end. ❷ Variable tags are used to either define variables or print the values of the variables. Within a variable tag, most TEs support *filter*s to manipulate the variables flexibly. For example, Smarty provides an *upper* filter to capitalize the value of a variable before printing, which looks like `{$a | upper}`. Common data types are supported in variable tags, e.g., numeric, string, and array. Except for the template-defined variables, the template code can access PHP variables that are propagated into the template and the TE-defined global variables. ❸ Function tags are used to invoke TE-defined functions or template-defined functions. Similar to a PHP function, the function tags can be invoked with parameters, such as `{func param1='val1' param2='val2'}`.

According to the template rendering workflow, TEs can be divided into two types: interpretation-based and generation-

[1] https://www.smarty.net/
[2] https://twig.symfony.com/

based. Interpretation-based TEs parse a template and directly render the HTML content by interpreting its tags. For every rendering request to the template, interpretation-based TEs always need to parse the template and interpret the logic of template tags. Hence, the whole rendering process is time-consuming and introduces negligible cost. By contrast, given a template, generation-based TEs parse its syntax and translate the template (including the logic of its tags) into a PHP file. Then, the translated PHP file is executed to generate the final HTML content. By caching the translated PHP file, all the subsequent rendering requests to the template are handled by executing the PHP file. Besides, once a template is modified, TE would automatically update the cached PHP file through re-translation. In this way, generation-based TEs achieve superior performance than interpretation-based TEs. Therefore, most popular TEs are generation-based.

2.2 SSTI and TE Sandbox

During the template rendering process, template code is either directly executed by a TE through interpretation or indirectly executed by running the translated PHP code. Such capability opens a new injection vector for attackers, known as server-side template injection (SSTI). An SSTI attacker is assumed to have control over the code of a template. By triggering the rendering of the template, the attacker can execute any template code on the server side.

Since TEs usually provide the template code with rich functionalities, SSTI attacks lead to severe consequences. The most severe exploit primitive enabled by an SSTI attack is remote code execution (RCE). For example, Smarty allows a function tag to invoke the `system()` PHP function, which could run any shell command, e.g., `{system("ls")}`. Besides, Smarty even provides a `{php}` tag to run PHP code in the template. Other popular TEs also provide the template code with similar capabilities. By leveraging these TE-supported capabilities, an SSTI attacker could invoke sensitive PHP functions and even run arbitrary PHP code on the server side, achieving an RCE exploit primitive. Moreover, diversified tags in TEs also enable SSTI attacks to achieve other exploit primitives, e.g., Local File Include (LFI) and Cross-Site Scripting (XSS) [26].

To defeat SSTI attacks, TE developers have designed some mitigation mechanisms. The common idea is to introduce a sandbox mode to restrict the capabilities of the tags provided with a template. For example, Smarty sandbox mode disables dangerous tags such as `{php}` and sets an allowlist of callable PHP functions from the template. Although some vulnerabilities have been reported in the TE sandbox due to the incomplete implementation (e.g., CVE-2014-8350, CVE-2015-7809), the TE sandbox mode is quite effective in mitigating SSTI attacks. Under the sandbox mitigation mechanism, attackers become hard to achieve RCE through SSTI. Hence, sandbox mode is frequently recommended as a countermeasure for SSTI attacks [1].

2.3 Sandbox Bypass: Template Escape Bugs

In this paper, we study an overlooked TE bug that could empower an SSTI attack to bypass the sandbox and gain RCE again. We use CVE-2021-26120 as an example to illustrate such kind of bug. Figure 1 (a) shows a code piece of Smarty that translates the `{function}` tag in a template into PHP code. At Line 1, the function name defined in the `{function}` tag (i.e., the `$_name` variable) is used to compose the corresponding PHP function name. Later at Line 6, the composed PHP function name is directly output to the PHP file, including other translated lines. By examining the code piece in Figure 1 (a), we observe that the `$_name` variable at Line 1 can be controlled by an SSTI attacker, and its value is directly output to the PHP file at Line 6 without any sanitization. Such code logic puts the translated PHP code at the risk of being manipulated. However, a normal `$_name` can not gain execution as PHP code. To gain code execution through the controlled template function name, an attacker should *escape the template semantic of using it as a part of a PHP function name*.

By carefully crafting the template function name, Figure 1 (b) shows a payload that successfully exploits the weakness at Line 1 of Figure 1 (a) to inject PHP code into the translated PHP file and gains execution. The translated corresponding PHP file is shown in Figure 1 (c), and the injected PHP code is `system("id")` at Line 1. We observe that there are three important parts in the malformed template function name, i.e., "`name(){};system("id");function `". First, the "`name(){};`" part is used to close the definition of the corresponding PHP function for the `function` tag. Second, the "`system("id")`" part is the real injected PHP code. Third, the "`function `" part is used to fix the incomplete implementation code of the corresponding PHP function for the `function` tag. In summary, the bug of the unsanitized input at Line 1 of Figure 1 (a) could only be exploited to enable an RCE attack by concatenating all these three parts.

Based on the above analysis, we conclude that the root cause of the bug in Figure 1 (a) is that *the attacker-controlled inputs in the template code escape the template semantic during the translation to PHP code*. Hence, we name such a bug as template escape. With the capability of bypassing the TE sandbox and elevating SSTI attacks to RCE, template escape bugs are considered to be severe. However, after a thorough examination of NVD, we find only two known template escape bugs (i.e., CVE-2017-1000480 and CVE-2021-26120), and both are reported in Smarty. Since the template translation process covers both the template code parsing and the PHP code generation, which involves lots of complicated string operations, we suspect template escape bugs should not be so rare in the wild. These facts motivate us to study the prevalence and the severity of template escape bugs in real-world TEs.

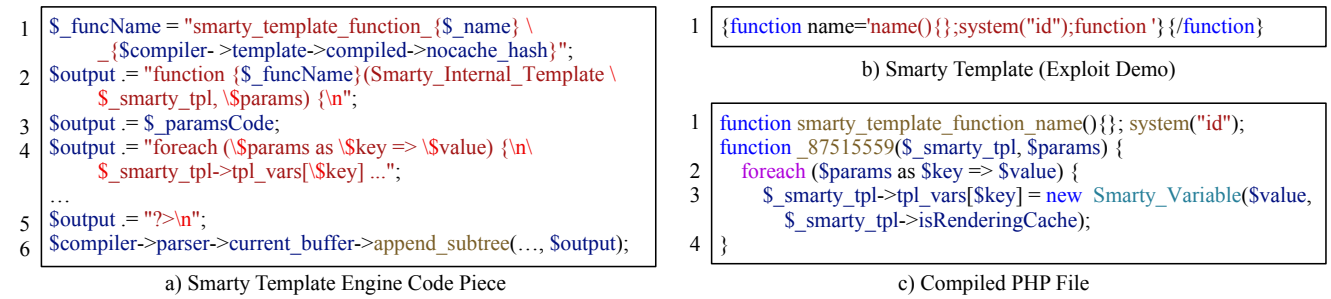

Figure 1: A Template Escape Bug in Smarty (CVE-2021-26120).

3 Overview

To study the prevalence and the severity of template escape bugs, we need an automated tool that could give a PoC to confirm a template escape bug and give an exploit to assess its exploitability. In the following, we will clarify the threat mode of template escape bugs, elaborate on the challenges of building such a tool, and give an overview of our approach.

3.1 Threat Model

To exploit template escape bugs, we assume that an attacker can inject template code, i.e., having an SSTI vulnerability. We observe that attackers can gain SSTI capability in (at least) three ways:

1. *Direct Template Code Injection.* It is common to accept external inputs during template rendering, making template injection a pervasive threat to web applications [4, 7].
2. *Exploiting Other Types of Vulnerabilities.* For example, template injection can be achieved by exploiting a file upload vulnerability which cannot upload PHP files but may support overwriting a template file [6, 8].
3. *Abusing Some Normal Functionalities of Web Applications.* For example, WordPress, Drupal, and OctoberCMS support UI customization through template modification; Cachet allows users to create customized reports by defining new templates.

Based on the above analysis, we conclude that SSTI attacks are hard to avoid in the real world. However, even with the capability of template code injection, an attacker cannot gain RCE, due to the TE sandbox. That is why template escape bugs are considered to be a type of severe vulnerability, for the capability of turning an SSTI attack into an RCE attack.

3.2 Challenges

In essence, the internal logic of a TE is quite complicated. On the one hand, it needs to parse a template file according to its specific syntax. On the other hand, it requires generating well-formed PHP code. Both steps involve massive string-related operations. Therefore, as illustrated in §2.3, it is pretty subtle to trigger and exploit template escape bugs. The complicated nature of a TE also brings vast challenges to the automated detection and exploitation of such bugs. We summarize the following challenges.

Challenge-I: It requires a fine-grained analysis of the template input for a TE. Though an attacker could control the template input, only a part of the input will be output to the PHP file, e.g., the name of a self-defined template function at Line 1 of Figure 1 (b). Therefore, detecting template escape bugs requires a fine-grained analysis to identify the structured elements of template input. However, since different TEs have their specific grammar, it is hard to learn the syntax of the template input.

Challenge-II: It requires a specific payload to trigger and exploit a template escape bug. As shown in Figure 1 (a), the PHP code generation process involves a lot of string-related operations (e.g., checks and transformations). Only carefully-constructed payloads could trigger a template escape bug. Furthermore, synthesizing an exploit is also quite challenging. Taking Figure 1 (b) as an example, successful exploitation requires carefully adapting the payload to the code around the escaped point in the PHP file.

Challenge-III: There lacks an oracle for identifying template escape bugs and successful exploitation. Unlike memory corruption bugs which can be identified by sanitizers (e.g., ASAN [47], UBSAN [11], and KMSAN [9]), the automatic detection and the exploitation of template escape bugs still lack an oracle. Specifically, no matter an input escapes the template semantic or not, there is no reliable indicator. Moreover, even if an input successfully exploits a template escape bug, it is hard to judge whether the generated PHP file has been injected with executable code.

3.3 Approach Overview

In light of these challenges, we propose a dynamic testing-based approach, called TEFUZZ. The basic idea is to set up a testing framework for different TEs and create testcases to discover and exploit template escape bugs. The advantages of dynamic testing are two-fold. First, it avoids complicated code analysis in understanding the syntax of template code and reasoning the control-/data-flow constraints. Second, it could leverage the generated PHP file to facilitate the identification of successful template escapes and exploitation.

However, TEFUZZ also meets the fundamental challenges of being a testing-based framework: *How to increase the testing coverage and avoid redundant testing*. Inspired by the research progress in fuzzing, we adopt several classic

principles and propose some new designs in TEFuzz to make it an effective, tailored fuzzing framework for template escape bugs. Before elaborating on the approach, we first define several key terms to ease the presentation.

Testcase. In TEFuzz, a testcase is a piece of code in the format of the tested TE. TEFuzz detects template escape bugs by testing and mutating testcases.

Escape Point (EP). As described in §2.3, template escape bugs occur when attacker-controlled inputs in the template code escape the template semantic during the translation to PHP code. However, only a part of the template code will be output to the PHP file, thus having the potential to escape template semantics. We use an *escape point (EP)* to refer to such a part of a template testcase. For example, the `$_name` variable of the `{function}` tag in Figure 1 (b) is an *EP* of the testcase.

Interesting Testcase. According to the root cause of template escape bugs, only testcases with *EP*s have the potential to trigger template escape bugs. We denote a testcase as an *interesting testcase* if it contains at least one *EP*.

Escape Context (EC). The triggering and exploitation of template escape bugs require the malformed template code at the *EP* to escape its original semantics in the translated PHP file. We refer to the code context of the *EP* in the output PHP file as the *escape context (EC)* of an interesting testcase. For example, the *EC* in Figure 1 (c) is the function name of the PHP function definition statement.

3.3.1 How to detect template escape bugs?

Principle-I: Balancing Exploration and Exploitation.[3] Finding a balance between vulnerability exploration and exploitation is a well-received principle in fuzzing. However, the way to balance exploration and exploitation is different and specific to the bug type. For example, binary fuzzing favors new code blocks in the exploration stage but favors new hits of covered code blocks in the exploitation stage [46]. According to the root cause of template escape bugs, we organize the two fuzzing stages as follows: the exploration stage aims to identify all interesting testcases; the exploitation stage looks for the right payload to turn an interesting testcase into a PoC. For each stage, we also propose a new technique to make it more effective.

❶ *Probing-based Interesting Testcase Identification.* Since a template testcase may contain many characters, it is challenging to efficiently infer the potential *EP* from a testcase (if it has). To this end, we propose a testcase probing technique to automatically identify interesting testcases, including their *EP*s. The high-level idea is inspired by probing-based input type inference [58] and challenge-based injection point identification [27]. Technically, we insert a magic string [4] into every position of the template code to gather a set of new testcases. By collecting the generated PHP file for each new testcase, we could use the magic string to infer which position of the given testcase will be output to the PHP code.

❷ *PHP Syntax-Guided PoC Generation.* To find the right payload to trigger the template escape bug at an *EP*, a naive mutation strategy would cause a vast testing space to explore. Our observation is that when a payload triggers a template escape bug, it also breaks the original code structure of the generated PHP file. More specifically, with the aim to break the PHP code structure, the payload should contain some syntax characters of the PHP language (e.g., '`;`' and '`)`'). Thus, we could collect all these characters and use them to generate PoC testcases. Moreover, by observing the code structure of the generated PHP file, we could effectively identify a successful template escape (i.e., a bug oracle).

Principle-II: Improving Code Coverage while Avoiding Redundant Testing. Covering more code is useful for bug detection, while redundant testing on similar testcases hurts the fuzzing efficiency. Following this principle, we propose two techniques to make the fuzzing process more effective.

❶ *Testcase Adaption by Leveraging Error Feedback.* To increase the testing coverage, TEFuzz would generate a lot of new testcases. However, the newly-generated testcases might meet various testing errors, due to dependencies on the running environment, grammar issues, etc. Therefore, we propose actively adapting the failed testcases to the testing target. Our observation is that the error feedback given by a TE usually conveys helpful information about how to fix the failed testcases. Considering that a TE usually has few types of error feedback, we introduce a semi-automated testcase adaption technique by leveraging the TE feedback. This technique substantially enhances the bug detection capability.

❷ *Testcase Clustering by Leveraging Runtime Information.* In the two-stage fuzzing, TEFuzz would inevitably generate similar testcases. First, the exploration stage would identify a lot of similar interesting testcases, because the *EP* inference is performed at every position of the testcase. Second, the exploitation stage would find similar PoCs with the same bug root cause. To cluster similar testcases, existing fuzzing frameworks have devised some metrics, such as code coverage [2, 39] and stack trace [37, 57]. In TEFuzz, we also propose useful metrics to cluster similar testcases at each stage by leveraging the runtime information of testcases.

3.3.2 How to exploit template escape bugs?

Based on a PoC that triggers a template escape bug, TEFuzz further tries to generate an RCE exploit. After escaping the template semantic, the malformed PoC also breaks the original code structure of the generated PHP file, which cannot execute normally. Thus, we need to adjust the PoC so that it not only breaks the original PHP code structure but also keeps it well-formed. The key challenge is that a PoC may break the PHP code structure in different contexts (aka *EC*), and each broken PHP code structure has to be fixed in its specific way.

[3] The exploitation here means finding a PoC rather than an exploit.
[4] https://en.wikipedia.org/wiki/Magic_string

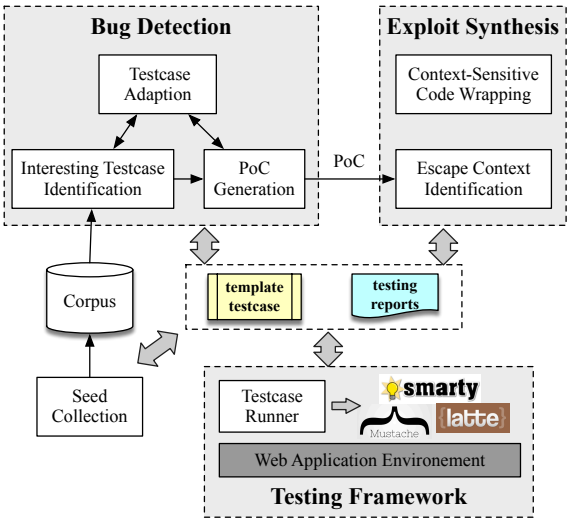

Figure 2: Overall Architecture of the TEFuzz Framework.

Hence, we propose context-sensitive exploit synthesis. Technically, there are two steps: ❶ identifying the *EC* of the PoC in the PHP code and ❷ adjusting the PoC to keep the injected PHP code fit the corresponding context. Note that existing works on XSS payload generation also meet a similar challenge, i.e., the injected payloads should fit various client-side contexts (e.g., HTML, JavaScript) [27, 29] and adopt a similar idea. Compared with existing works, our technical contributions are the summarized *EC*s for template escape bugs and the payload adjustment methods to fit PHP code.

4 Detailed Design

In this section, we present the detailed design of our approach. Figure 2 shows the architecture of TEFuzz, which consists of four key modules. The *Testing Framework* module setups a testing environment for each TE. It takes a template file as input and returns a testing report for each testcase. The *Seed Collection* module collects a set of template files as the seeds for generating testcases, PoCs, and exploits. The *Bug Detection* module mutates a seed testcase to discover template escape bugs. When a bug is triggered, the testcase is collected as the PoC. The *Exploit Synthesis* module takes a PoC as input and weaponizes it as an exploit for the target bug. Algorithm 1 depicts the overall workflow. During the whole workflow, all the testcases (including PoCs and exploits) are executed in the testing framework. In all, there are four steps.

- *Step 1: Seed Collection (Line 2)*. We collect an initial set of testcases for each TE. The details are described in §5.2.
- *Step 2: Interesting Testcase Identification (Lines 3-14)*. For each seed testcase, we propose a probing-based technique (§4.2.1) to create many testcases and identify which are interesting ones. If some testcases run abnormally, we use the error feedback to adapt these testcases (§4.2.3). We also cluster the interested testcases to avoid redundant testing.

Algorithm 1: TEFuzz Workflow

```
   Input  : TE
   Output : PoCs, Exploits
1  interest_testcases ← []; PoCs ← []; Exploits ← [];
2  seeds ← SeedCollection ();
3  for testcase in seeds do
4      new_testcases ← Probing(testcase);
5      for testcase in new_testcases do
6          report ← ExecInFramework(testcase);
7          if IsTEError(report) then
8              testcase ← TestcaseAdaption(testcase);
9              report ← ExecInFramework(testcase);
10         if IsInteresting (report) then
11             Add(interesting_testcases,testcase);
12     end
13 end
14 interest_testcases ← Clustering1(interest_testcases);
15 for testcase in interest_testcases do
16     new_testcases ← Mutate(testcase);
17     for testcase in new_testcases do
18         report ← ExecInFramework(testcase);
19         if IsTEError(report) then
20             testcase ← TestcaseAdaption(testcase);
21             report ← ExecInFramework(testcase);
22         if IsPoC (report) then
23             Add(PoCs, testcase);
24             break;
25     end
26 end
27 PoCs ← Clustering2(PoCs);
28 for poc in PoCs do
29     context ← ClassifyContext(poc);
30     exploit ← CodeWrapping(poc, context);
31     report ← ExecInFramework(exploit);
32     if IsExploit(report) then
33         Add(Exploits, exploit);
34 end
```

- *Step 3: PoC Generation (Lines 15-27)*. For each interesting testcase, we propose PHP syntax-guided testcase mutation technique (§4.2.2) to create many testcases and identify which ones can trigger template escape bugs (i.e., are PoCs). For the error testcases, we also use the testcase adaption technique (§4.2.3) to adapt them. We further cluster the PoCs to remove duplicated bugs.
- *Step 4: Exploit Synthesis (Lines 28-34)*. For a given PoC, we first identify its *EC* in the generated PHP file (§4.3.1) and then adjust the PoC to wrap the PHP code around the *EC* (§4.3.2).

4.1 Testing Framework

Unlike traditional user-space programs, the testing of a TE has various dependencies on the web environment. For example, TEs usually access some global variables defined by the

web platform (e.g., `$_SERVER`). Hence, we set up a complete PHP web application environment in the testing framework. Further, upon the web application environment, we build a testcase runner who runs a template testcase on a target TE and collects the feedback to facilitate bug detection and exploit synthesis. Since TEs have different interfaces, we create a separate driver for each TE. The logic of a TE driver is quite simple. It just invokes the target TE to render the given template testcase.

For each testcase, the testing framework returns a testing report. First, it collects the errors that occur during the testing. The errors might be caused by two sources: ❶ *TE errors* that happen during the template translation; ❷ *PHP errors* that occur during the execution of the generated PHP file. Second, it collects the covered code lines of a TE during the testing. As it will be described later, the coverage information helps avoid redundant testing. Third, it collects the generated PHP files, which are used to identify escape points, synthesize exploits, etc. Generally, the testing framework interacts with other modules by receiving testcases and sending testing reports.

4.2 Bug Detection

4.2.1 Interesting Testcase Identification

We collect interesting testcases by identifying the escape points (*EP*s) of each seed testcase. To avoid understanding the complicated syntax of the template input, we adopt a probing-based technique. There are mainly four steps (Lines 3-13 in Algorithm 1):

1. We insert a magic string (e.g., "Un1QuE") into each position of the seed testcase and get a set of new testcases.
2. We send each new testcase to the *Testing Framework* module to get its testing report. If there is a TE error reported during the testing, we send the testcase to the *Testcase Adaption* module to try to fix the error. If the TE error cannot be fixed, we discard the testcase.
3. We extract the generated PHP file for each testcase from its testing report and match the inserted magic string in the PHP file. If there is no match, we discard the testcase because there is no *EP*.
4. We collect all the testcases that have matched magic strings in their generated PHP files, including their testing reports. These testcases are identified as interesting testcases.

Interesting Testcase Clustering. Since the identification of interesting testcases is based on inference, we may identify many similar interesting testcases. Let us consider the `function` tag in Smarty as an example, as shown in Figure 1 (b). When we insert a magic string before the function name's first and last character, we may get two interesting testcases. However, they both belong to the same *EP* (i.e., the function name of the tag). Thus, we need to cluster similar interesting testcases to avoid redundant testing. The key challenge is that the clustering method should not rely on prior knowledge of the template syntax.

To this end, we propose a syntax-agnostic testcase clustering technique. Our insight is that similar interesting testcases should have the same code footprint during the template translation and the same *EP*s in the generated PHP files. Specifically, we consider two testcases as different only when they have different code coverage or *EP*s. The code coverage of a testcase can be directly acquired from the testing report. Meanwhile, the *EP*s in the generated PHP file are represented by the line numbers of the magic string in the PHP file. According to the evaluation (§5.7), the interesting testcase clustering technique significantly helps to reduce a large number of redundant testing.

4.2.2 PoC Generation

Based on an interesting testcase, we aim to find the right payload to trigger a template escape at its *EP*s. Our observation is that if we expect the payload to escape the template semantic, the payload should contain some pre-defined PHP syntax characters, e.g., '`;`' and '`)`'. Therefore, we propose syntax-guided testcase mutation. It works in the following four steps (Lines 15-26 in Algorithm 1):

1. We collect all the pre-defined PHP syntax characters by reading the specifications. Besides, we also include some other common escape characters, such as "`*/`". In all, there are 134 escape characters to test.
2. We insert these escape characters into each *EP* of an interesting testcase to create new testcases.
3. We run each new testcase in the *Testing Framework* module. We also use the *Testcase Adaption* module to fix the TE errors reported during the testing.
4. We identify a PoC by using the generated PHP file as an oracle. In particular, if the generated PHP file reports a PHP error during the template rendering or its original code structure changes, we consider the testcase as a PoC.

PoC Clustering. Like other fuzzing-based bug detectors, TEFUZZ may report duplicate PoCs with the same root cause [23, 52]. For example, `{block "*/title"}{/block}` and `{block name="*/title" prepend}{/block}` are two PoCs but have the same root cause: the `name` attribute of the `block` tag escapes the template semantic. To reduce the manual efforts in bug validation, we remove duplicate bugs by PoC clustering.

Inspired by the interesting testcase clustering technique, we cluster PoCs based on their code coverage and *EP*s. Specifically, given two PoCs, we first measure a coverage similarity using the Jaccard index [5] between their covered code line numbers. Then, we measure an *EP* similarity by using the edit distance [6] between the PHP code lines that include their *EP*s. To avoid identifying two unique bugs as the same, we adopt a conservative design. On the one hand, we set a high threshold (i.e., 0.95) for both the coverage similarity

[5] https://en.wikipedia.org/wiki/Jaccard_index
[6] https://en.wikipedia.org/wiki/Edit_distance

and the *EP* similarity. On the other hand, we identify two PoCs as the same only when both the coverage similarity and the *EP* similarity exceed the thresholds.

4.2.3 Testcase Adaption

Both the *Testcase Probing* module and the *PoC Generation* module create new testcases by mutation. These mutated testcases may meet various errors during the testing. We try to fix the failed testcases to improve the testing coverage. However, fixing testcases is quite challenging due to the semantic constraints on a valid testcase. Our observation is that the error messages given by a TE for failed testcases usually convey helpful information for fixing. Therefore, we design feedback-guided testcase adaption. For each TE, we manually collect the error messages that it may report during the testing and compile the corresponding adaption rules to fix the testcase. Fortunately, we find that the number of such error messages is not large, which makes the manual analysis affordable. According to our experience, we summarize three types of TE errors: *file error*s, *grammar error*s, and *attribute error*s. These errors are fixed in the following ways.

- *File Adaption*. A template testcase usually refers to some files, while access to these files may cause errors. There are two scenarios. First, the required file does not exist or is not readable. For this error, we will create this file and give it read permission. Second, if the included file directory is not trusted by a TE, we will move the file to a trusted directory. For example, when an error message says "*unable to load template* `test1.tpl`", we will create a "`test1.tpl`" file in the corresponding directory.
- *Grammar Adaption*. The mutated testcases frequently meet grammar errors. We find three categories of such errors: i) missing parameters or attributes, ii) unclosed symbols, and iii) unclosed tags. Accordingly, we match these errors and try to fix them. For example, when we meet an error saying *"unclosed* `{block}` *tag"*, we will add `{/block}` at the end of the seed to fix the unclosed tag error.
- *Attribute Adaption*. The mutation may break the values of some tag attributes, making their type unsatisfied. For such errors, we will change the values until they become legal. For example, when an error message says *"illegal value for option attribute* `inline`*"*, we will change the value of the `inline` attribute into another type.

These adaptions help run more testcases during the testing, which ultimately helps discover more bugs. Note that when a single round of adaption fails to fix a testcase, we would repeat the adaption until the total round of adaptions reaches a given limit (e.g., 2 in our current setting).

4.3 Exploit Synthesis

4.3.1 Escape Context Identification

***EC* Taxonomy.** According to the output position of an *EP* in the generated PHP file, we summarize five types of *EC*s.

- *Function/class definition.* When defining some PHP functions or classes, TEs may name them using template code elements.
- *Function invocation statements.* The template code elements may be used to refer to the PHP function or its parameters at function invocation statements.
- *Assignment statements.* TEs usually use template code elements to generate assignment statements, e.g., when translating `variable` tags.
- *Conditional statements.* The variables defined in the template code may appear in some conditional statements of a generated PHP file, e.g., `if` statements, `foreach` statements.
- *Comments.* To increase the readability of the generated PHP file, TEs usually generate comments in the PHP file using some elements of the template code.

Based on the above taxonomy, we identify the *EC* of a PoC by locating its *EP* in the abstract syntax tree (AST) of the generated PHP file. Note that when a PoC triggers a template escape bug, it breaks the proper code structure of the PHP file, making locating the *EP* very hard. Therefore, we use a benign input to replace the malformed payload of the PoC and then generate a valid PHP file. By locating the node of the benign input in the valid AST, we identify the PHP statement that this node belongs to and the *EC* based on this PHP statement.

4.3.2 Context-Sensitive Code Wrapping

According to the specific *EC*, we adjust the PoC to inject malicious PHP code into the generated PHP file. The most difficult part is to keep the generated PHP file well-formed, which requires finding the right payload to wrap the PHP code before and after the *EP*. Our idea is to use the *EC* taxonomy in §4.3.1 as guidance and adopt a template-based approach to generate the right wrapping payload for each PoC. Specifically, there are three steps.

1. According to the *EC* and the *EP*, we look for the right payload to close the current PHP statement. The overall strategy is like a switch case, i.e., using the *EC* and *EP* as the key to locate the corresponding template for payload generation. For example, if the *EC* is a PHP function definition statement and the *EP* is the function name (see Figure 1), we can use the payload "`name(){};`" to close the function definition statement For another example, if the *EC* is a PHP function invocation statement and the *EP* is the parameter, we can use "`));`" to close the function invocation statement (the number of the right parentheses depends on the number of the left parentheses in the *EC*).
2. After wrapping the code before the *EP*, we locate the AST nodes for the code after the *EP*. A straightforward way to close these nodes is to put them into a comment. For example, in most cases, we can use the payload "`//`" to wrap the code after the *EP*. However, a line comment could not wrap all the code when the remaining code spans multiple lines. In this case, we adopt the same strategy as

the previous step to seek a payload that could turn the remaining code into a valid PHP statement. Specifically, we define several payload generation templates according to the *EC* and *EP* of the current PHP statement. For example, if the *EC* is a PHP function definition statement, we can use "`function `" to close the remaining code (see Figure 1).

3. Using the above two payloads to wrap the code around the *EC*, we insert the target RCE code between the payloads. For example, we simply use "`system('id');`" as the RCE payload to demonstrate the success of the exploit.

Following these steps, we synthesize an exploit from a PoC. We further run the synthesized exploit in the testing framework to verify its validity.

5 Evaluation

5.1 TEFuzz Prototype

We implemented a prototype of TEFuzz for PHP TEs, which consists of 4,300 lines of Python code and 560 lines of PHP code. The *Seed Collection* module is built with the Python *requests* package. The *Testing Framework* module is set up on an Apache web server with PHP version 7.2.34 to provide a running environment for the tested TEs. It communicates with other modules via HTTP, i.e., a template testcase is sent to the testing framework as an HTTP request, and the testing report is returned within the HTTP response. During the test, Xdebug [7] is used to collect the code coverage of the tested TE. Both the *Bug Detection* module and the *Exploit Synthesis* module are implemented in Python. The two modules execute in the way by sending testcases to the *Testing Framework* module and analyzing the returned testing reports.

5.2 Experimental Setup

TE Dataset. Our study requires collecting a TE dataset. In particular, we set three criteria to select TEs. First, we only consider generation-based TEs. Second, it should be a popular TE. Third, the TE should have a sandbox mode to mitigate SSTI attacks (e.g., preventing RCE) or should not support executing PHP code in the template.

More specifically, we first used keywords (e.g., template engine) to search all the TE candidates on GitHub. We set the language to PHP and only collected the TEs with more than 100 GitHub stars. Following this way, we constructed a list of 18 TEs. Then, we filtered those TEs that are not generation-based (5 TEs) and those TEs that support direct PHP code execution in the template and do not have a sandbox mode (6 TEs). At last, we collected 7 PHP TEs that satisfy all the above three criteria. As listed in Table 1, our study is conducted on these TEs. Note that the latest versions of these TEs are used in the dataset.

We give a short description of these TEs. In our dataset, Smarty, Latte, Dwoo, and Fenom all provide some tags for PHP code execution and designs a sandbox mode to restrict the usage of these sensitive tags. Though Twig does not introduce a specific tag for PHP code execution, developers could run arbitrary PHP code using some filters, such as `map` and `sort`. The sandbox mode of Twig is designed to restrict the usage of these filters in the `variable` tags. Unlike other TEs, Mustache is a simple, logic-less TE, which inherently does not support PHP code execution nor support invoking PHP functions in the template, so an attacker can not gain RCE by controlling a Mustache template. Owing to its lightweight design, Mustache is extremely popular. ThinkPHP is well-known as a PHP application framework; however, its internal template engine has also been widely used by other web applications. Therefore, we also include its TE in the dataset. ThinkPHP TE provides some tags to support PHP code execution, and its sandbox mode limits the execution of PHP code by sanitizing the parameters to these tags through "TMPL_DENY_PHP", and "tpl_deny_func_list".

We manually make the following adaptions to apply TEFuzz to the collected TEs. First, we manually created a driver for each TE to support testing its template testcases in the *Testing Framework* module. The testcase driver is easy to develop and usually requires less than 50 lines of PHP code. To discover bugs that could escape the TE sandbox, the sandbox mode of the TE is also enabled during the testing. Second, we extracted the delimiters that each TE uses from the documentation. As listed in Table 1, we find that though TEs use different delimiters for `function` tags, `variable` tags, and `comment` tags, the numbers of delimiters are small. These delimiters are used in the *Seed Collection* module and the *Testcase Adaption* module to facilitate the template seed collection and the template error fixing, respectively. Third, according to the error messages that are reported during the testing, we manually compiled adaption rules in the *Testcase Adaption* module to fix these errors. On average, 5.4 adaption rules are defined for a TE (see Table 1).

Seed Collection. The seeds of each TE are collected in two steps. First, for each TE, we use crawlers to collect its official documents and the testing files in its source code. Second, we write regular expressions based on the delimiters to extract the template code from the crawled texts. In this way, we collected an initial corpus of 2,527 template testcases. We run these testcases in the testing framework and find that 1,621 ones can run normally. For the 906 cases that report errors, we used the *Testcase Adaption* module to fix them, which successfully fixed 107 cases. At last, we collected a set of 1,728 testcases as the initial seeds for TEFuzz. Table 1 shows the number of collected seeds for each TE. In all, the whole process of seed collection costs about 3.5 man-hours.

Experiments. In the following, our experiments are organized by answering the following questions.

- RQ1: How prevalent are template escape bugs?
- RQ2: How severe are template escape bugs?

[7] https://xdebug.org/

Table 1: Dataset of the Target TEs and Their Basic Information.

TE Name	Version	Stars	LoC	Mitigation	Delimeter	# of Seeds	# of Adaption Rules
Smarty	v3.1.39	2k	25,986	Sandbox	{,}; {*,*}	523	13
Twig	v3.3.1	7.5k	18,378	Sandbox	{{,}}; {%,%}; {#,#}	339	9
Dwoo	v1.3.7	168	80,405	Sandbox	{,}; {*,*}	208	4
Latte	v2.10.5	802	6,949	Sandbox	{,}; {*,*}	289	5
Mustache	v2.14.0	3.1k	6895	No PHP Execution	{{,}}; {{!,}}	17	2
Fenom	v2.12.1	431	11,974	Sandbox	{,}; {*,*}	181	4
ThinkPHP	v6.0.12	2.4k	2,280	Sandbox	{,}; {/,/}	171	1

Table 2: Detected Bugs (RQ1 & RQ2).

TE Name	Unique Bugs	Exploitable Bugs	RCE?
Smarty	3	3	✓
Twig	0	0	
Latte	49	24	✓
Mustache	1	1	✓
Dwoo	38	2	✓
Fenom	10	10	✓
ThinkPHP	34	15	✓
All	135	55	

- RQ3: How does TEFuzz compare to SSTI scanners?
- RQ4: How feasible is exploiting template escape bugs in real-world applications?
- RQ5: How helpful are the internal designs of TEFuzz?

5.3 Prevalence of Template Escape Bugs (RQ1)

We have manually analyzed the reported PoCs by following their execution traces and pinned the code lines that cause these bugs. We deem two bugs as the same if they are caused by the same (buggy) code lines. Table 2 presents the bug detection results and Column 2 shows the number of bugs detected in each TE. From Column 2, we find that almost every TE has template escape bugs. In all, TEFuzz reports 135 bugs in these TEs. The results show that template escape bugs are quite prevalent.

From Table 2, we also observe that the number of bugs varies significantly among different TEs. For example, Latte has 49 bugs, while Twig reports no escape bugs. Therefore, we further investigate the following two questions: ❶ What are the root causes of these bugs? ❷ Why do some TEs have fewer bugs? With the help of the generated PoCs, we manually examined all the code that uses and checks the malformed inputs. Technically, we find the template translation process can be divided into two continuous steps: *template code parsing* and *PHP code generation*. In each step, improper input validation would cause template escape bugs. We present the manual analysis results below.

Root Cause. In general, we find two causes for these bugs.

- *Incomplete validation in template code parsing (120 bugs).* During the parsing of the template code, TEs need to validate whether the template code elements fit the template syntax. For example, the name of a template variable should only include digits, letters, and underscores. However, we observe that some bugs are caused because the TEs do not perform a complete validation on the template code elements. Taking Latte as an example, it accepts special symbols (e.g., '/' and ';') as a part of a variable name during the parsing. Though these bugs do not affect the normal functionality of template parsing, they can be used to inject malformed payloads into the template translation.
- *Incomplete sanitization in PHP code generation (15 bugs).* After parsing the template code, the template code elements will be used to generate different PHP code statements. Even when these elements have been validated to fit the template syntax, TEs still need to sanitize them to prevent the PHP syntax from escaping. However, we observe that some TEs do not properly perform all the necessary sanitization, causing template escape bugs. For example, the Dwoo template code "`{assign bar foo}`" is used to assign the `foo` variable with a constant string "`bar`". During the code generation, the template code is translated into a PHP function invocation statement, and the constant string "`bar`" is used as a parameter. However, though Dwoo has sanitized some characters (e.g., '`(`') for the constant string attribute, we find that it forgets to sanitize the character '`\`'. Therefore, TEFuzz reports an escape bug in this case.

Different Practice among TEs. In all, we find four features of the TEs that are more prone to template escape bugs.

- *Complex template syntax.* Mustache has the most simple syntax in our TE dataset and the smallest number of tags. Therefore, we only find one bug in it.
- *Directly mapping the template variables to PHP variables.* Defining variables in the template code is supported in every TE. Some TEs directly map these variables to PHP variables with the same name during the template code translation. Such practice allows attackers to use malformed template variable names to escape the PHP syntax. We find that both Latte and ThinkPHP follow this practice and report many bugs due to this reason.
- *Optimistic template code parsing.* Some TEs parse the template code in an optimistic way, e.g., Latte, Dwoo and ThinkPHP. That is, they try not to report syntax errors during the parsing. However, optimistic template parsing would make the PHP code generation prone to escape bugs. On the contrary, Smarty, Twig, Mustache, and Fenom all adopt strict syntax parsing, thus having fewer bugs.
- *Lacking sanitization on the generated PHP code.* When generating PHP code, TEs should sanitize the generated

```
1 {block name="*/system('id');//"}{/block}
```

(a) Smarty Tmplate File

```
1 /* {block "*/system('id');//"} */
2 class Block_19842409 extends Smarty_Internal_Block{
3     public $subBlocks = array (
4         '*/system(\'id\');//' =>  array (0 => 'Block_19842409',),
5     );
6     …
7 }
```

(b) Translated PHP File

Figure 3: A Template Escape Bug in the `{block}` Tag of Smarty (RQ2).

code according to its context in the PHP file. For example, if the generated code appears in the comments, the "`*/`" characters should be sanitized. Among all the TEs, we only observe Twig performing a context-aware code sanitization. As a result, TEFUZZ has not discovered a bug in Twig.

5.4 Severity of Template Escape Bugs (RQ2)

Based on the reported 135 template escape bugs, TEFUZZ successfully exploits 55 bugs (see Table 2). Right now, we have been assigned 4 CVEs. From Table 2, we also find that TEFUZZ successfully generates an exploit for every TE reported with a template bug. Among all the TEs in our dataset, only Twig has no template escape bug. For other TEs, an attacker could escalate a template modification vulnerability to RCE with the help of the synthesized exploits. We give a case study about the discovered exploitable bugs and the synthesized exploits.

Case Study: An Exploitable Bug in Smarty. TEFUZZ reports a bug in the `{block}` tag of Smarty (v3.1.39). As shown in Figure 3, when translating the `{block}` tag of a template, Smarty generates a PHP class for the block, including a brief comment to describe the class. From the generated PHP file, we can find that the name of the block directly appears in the generated comment. Using a "`*/`" in the block name, TEFUZZ successfully generates a testcase to escape the comment definition in the generated code, as shown in Figure 3 (b). When synthesizing the exploit, TEFUZZ further analyzes the comment escape context in the generated PHP file and adds some wrapping code to the block name to fit the escape context. At last, the synthesized exploit in Figure 3 (a) successfully injects the target RCE code into the generated PHP file, enabling an RCE attack.

5.5 Comparison with SSTI Scanners (RQ3)

To our knowledge, TEFUZZ is the first tool to detect and exploit template escape bugs. Nevertheless, we noticed an SSTI scanner, named *tplmap*, which claims to be capable of bypassing the TE sandbox. Thus, we conduct a comparison experiment between *tplmap* and TEFUZZ. Our experiment seeks to answer two questions: ❶ Can *tplmap* discover template escape bugs? ❷ Can TEFUZZ help *tplmap* to discover exploitable SSTI vulnerabilities?

Table 3: Comparison Results between *tplmap* and TEFUZZ.

TE Name	Version	*tplmap*			*tplmap* + TEFUZZ		
		SSTI	Escape[1]	RCE	SSTI	Escape[1]	RCE
Smarty	v3.1.39	✓	×	×	✓	✓	✓
Twig	v3.3.1	✓	×	×	✓	×	×
Dwoo	v1.3.7	✓	×	×	✓	✓	✓
Latte	v2.10.5	✓	×	×	✓	✓	✓
Mustache	v2.14.0	✓	×	×	✓	✓	✓
Fenom	v2.12.1	✓	×	×	✓	✓	✓
ThinkPHP	v6.0.12	✓	×	×	✓	✓	✓

[1] Triggering a template escape bug

Experiments. We use the TEs in Table 1 to test the *tplmap*. Since *tplmap* requires a standalone web application for scanning, we use the application drivers used to run the template testcases in the *Testing Framework* module as the entry points for *tplmap*. This setting eases the discovery of template injection points for *tplmap*. Besides, we find that *tplmap* only supports Smarty and Twig in our TE dataset. Therefore, we enhanced *tplmap* to support other TEs in our dataset (e.g., Mustache, Latte), by carefully following its internal mechanism.

In addition, according to our study on *tplmap*, we find that it relies on a set of manually-curated payloads to bypass the TE sandbox. Thus, we also use the RCE payloads that are automatically generated by TEFUZZ to enhance *tplmap* (when *tplmap* fails to break the sandbox).

Results. We present the results in Table 3. The results show that *tplmap* successfully discovers the template injection points in the application driver of each TE. However, due to the incapability of generating new sandbox bypassing payloads, *tplmap* fails to bypass the sandbox of any TE. In contrast, using the RCE payloads generated by TEFUZZ, *tplmap* successfully breaks the sandbox of every TE (except Twig). The results show that *tplmap* cannot discover template escape bugs, and TEFUZZ and *tplmap* are complementary in detecting exploitable SSTI vulnerabilities.

5.6 Feasibility of Full Exploitation (RQ4)

In the threat model of template escape bugs (§3.1), an attacker must gain the capability of template code injection. In this research question, we evaluate the feasibility of template code injection in the real world. Specifically, the experiments are conducted from two orthogonal directions. First, we assess the feasibility of SSTI attacks by looking into a database of known vulnerabilities. Second, we try to verify the full exploitability of template escape bugs by discovering some 0-day SSTI vulnerabilities.

5.6.1 Searching Known Vulnerabilities

By scrutinizing known SSTI vulnerabilities in the CVE database, we seek to answer the following two questions: ❶ What are the root causes of the SSTI vulnerabilities in the wild? ❷ Can known SSTI vulnerabilities be used to exploit the template escape bugs discovered by TEFUZZ?

Experiment-I: Root Causes of Known SSTI Vulnerabilities. First, we investigate the root causes of real-world template injection vulnerabilities to assess the feasibility of our threat model. To this end, we collect a set of template injection vulnerabilities by using the keywords "SSTI" and "template injection" to search the CVE database, and get 145 search results. We find that many of the searched vulnerabilities do not belong to template injection. Thus, we manually examine all the search results and eventually found 80 SSTI vulnerabilities. During the manual examination, we also pinpointed the root cause of each confirmed SST vulnerability according to the vulnerability description. In all, we find 45 vulnerabilities belong to *Direct Template Code Injection*, 8 vulnerabilities belong to *Exploiting Other Types of Vulnerabilities*, and 25 vulnerabilities belong to *Abusing Some Normal Functionalities of Web Applications*. Note that we fail to identify the root causes of 2 vulnerabilities due to the incomplete CVE description.

Experiment-II: Full Exploitation on Known SSTI Vulnerabilities. Second, we check whether the collected SSTI vulnerabilities can facilitate the discovered template escape bugs to achieve full RCE exploitation. We adopt two methods to confirm the used TE of an SSTI vulnerability: 1) examining the source code of the affected web application and 2) reading the description of the official website of the application. Following this way, we find three vulnerabilities that use a vulnerable TE in our dataset: two SSTI vulnerabilities that use Smarty (CVE-2020-35625 and CVE-2017-16783) and one vulnerability that uses ThinkPHP (CVE-2020-25967).

Note that though we only find three vulnerabilities that use a vulnerable TE in our dataset, there should be more cases in the real world. For example, by reading some security blogs and vulnerability reports, we also found two SSTI vulnerabilities (CVE-2017-6070 and CVE-2020-15906) that use Smarty, but have not been covered in the experiment. Therefore, we successfully collect five SSTI vulnerabilities that use a vulnerable TE in our dataset. With the help of the public vulnerability reports, we reproduced all these SSTI vulnerabilities. By using the synthesized RCE payloads by TEFuzz, we successfully conduct full RCE exploitation in the affected web applications. In the following, we give two case studies about the full exploitation.

Case 1. Tiki Wiki [8] is a free and open-source wiki-based content management system (CMS), which is built with Smarty. It has been reported with an authentication bypass vulnerability, i.e., CVE-2020-15906, which allows an attacker to reset the admin password. Since template modification is provided as a normal functionality, an attacker can run arbitrary template code on the server side by modifying the template. However, due to the sandbox mode of Smarty, the attacker is hard to run PHP code using the template modification capability. By leveraging the exploit synthesized for Smarty (e.g., Figure 3), we successfully run arbitrary PHP code at the vulnerable server. This case shows how a template escape bug elevates an authentication vulnerability to RCE.

Case 2. CMS Made Simple (CMSMS) [9] is an open-source CMS built with Smarty and widely used by developers and site owners. In CMSMS 1.X, there is an SSTI vulnerability (CVE-2017-6070) that allows an attacker to render arbitrary template content by using the "brp_fora_form_template" parameter, leading to an RCE attack [5]. This vulnerability lies in the Form Builder component of CMSMS, which helps users create feedback forms. Though the vulnerability affects both CMSMS 1.X and CMSMS 2.X, an RCE attack is only feasible on version 1.X (which may also depend on the configuration of the Smarty). The reason is that, on CMSMS 2.x, Smarty is used with the sandbox mode. As a result, an attacker is hard to exploit CVE-2017-6070 to achieve RCE on CMSMS 2.x. To verify the severity of the discovered template escape exploits in Smarty, we set up the exploitation environment for CMSMS 2.x. We find that though an attacker can inject arbitrary template code on the server side, it is hard to run arbitrary PHP code due to Smarty's sandbox mode. By replacing the injected template code with the exploit synthesized by TEFuzz (e.g., Figure 3), we successfully run "`system('id')`" on the server side. The results demonstrate the severity of template escape bugs that enlarge the attacking capability of an SSTI vulnerability.

Summary: We have two findings from the above experiments. ❶ The root causes of real-world SSTI vulnerabilities are consistent with the threat model in §3.1. That is, there are at least three ways to achieve SSTI, rendering it a real threat. ❷ By putting in great manual efforts (about 80 man-hours), we successfully verified the full exploitability of the discovered template escape bugs with *five* real-world SSTI vulnerabilities.

5.6.2 Discovering 0-day Vulnerabilities

In addition to searching for known SSTI vulnerabilities, we try to discover some 0-day SSTI vulnerabilities. To this end, we first collect a set of PHP applications and then try to discover SSTI vulnerabilities in these applications in two ways: SSTI scanners and manual hacking.

Application Dataset. We only collect the PHP applications that use the vulnerable TEs in our dataset. The collection process consists of three steps. First, we use keywords (e.g., "CMS") to search the projects in GitHub and only select the PHP applications with 500+ stars. In this step, we find 7 PHP applications that use the vulnerable TEs in our dataset. Second, we use some keywords (e.g., "Best Popular CMS 2022", "Best Popular PHP CMS", and "PHP CMS") to search popular PHP applications in Google. We initially got 55 applications from the search results and only kept nine applications that fit our requirements. Third, for those TEs

[8] https://tiki.org/

[9] http://www.cmsmadesimple.org/

Table 4: Details of the Collected 18 Real-world PHP Applications that Use TEs in Our Dataset.

CMS	TE	Version	SSTI Vulnerabilities		RCE	Stars	URL
			tplmap	Manual			
CMSMS	Smarty	2.2.16	×	✓	✓		http://www.cmsmadesimple.org/
lmxcms	Smarty	1.41	×	✓	✓		http://www.lmxcms.com/
Piwigo	Smarty	13.0.0	×	✓	✓	2.2k	https://github.com/Piwigo/Piwigo
UQCMS	Smarty	1.0.27	×				http://www.uqcms.com/
MediaWiki	Smarty	1.38.2	×	✓	✓	3.2k	https://github.com/wikimedia/mediawiki
alltube	Smarty	3.0.3	×			2.6k	https://github.com/Rudloff/alltube
pH7-Social-Dating	Smarty	17.2.0	×			807	https://github.com/pH7Software/pH7-Social-Dating-CMS
postfixadmin	Smarty	3.3.11	×			721	https://github.com/postfixadmin/postfixadmin
DouPHP	Smarty	1.7	×				https://www.douphp.com/
MODX	Smarty	3.0.1-pl	×				https://modx.com/
Tiki Wiki CMS	Smarty	21.7	×	✓	✓		https://tiki.org/HomePage
BlackCat	Dwoo	1.3.6	×				https://blackcat-cms.org/
tuleap	Mustache	13.9.99.33	×			811	https://github.com/Enalean/tuleap
Fansoro	Fenom	2.0.4	×			106	https://github.com/fansoro/fansoro
Fastadmin	ThinkPHP	1.3.4	×			1.5k	https://github.com/karsonzhang/fastadmin
feifeiCMS	ThinkPHP	4.3.201206	×				https://www.feifeicms.org/
74CMS	ThinkPHP	3.12.0	×				https://www.74cms.com/
ejucms	ThinkPHP	SP4	×	✓	✓		https://www.ejucms.com/

that have not been covered in the collected applications, we use their names as keywords to search for some specific PHP applications. Following the above steps, we collect a set of 18 PHP applications (as shown in Table 4). We find that the most number of applications have used Smarty. Besides, we do not find an application that uses Latte. According to the official website of Latte, we infer the reason is that Latte is mostly used by online websites [14], rather than open-source PHP applications. Next, we use the latest version of each application to set up a running environment for further vulnerability discovery.

Experiment-I: SSTI Scanners. Since *tplmap* is the only SSTI scanner we can find, we use it to discover SSTI vulnerabilities in the collected PHP applications. According to the documentation of *tplmap*, it should be given a set of URLs and parameters. Therefore, we use *crawlergo* [3, 62], a powerful crawler, to discover URLs in these applications, including the parameters, and then feed them to *tplmap* to test which URLs and parameters lead to template code injection. Overall, *crawlergo* costs 54 hours to discover 17,424 URLs and 41,940 parameters from these applications. However, after a total scanning period of 140 hours on these URLs and parameters, *tplmap* reports no SSTI vulnerabilities.

Given that there are many known SSTI vulnerabilities (see §5.6.1), we think the poor performance of *tplmap* renders a promising research direction to design a more effective SSTI scanner. According to our experience in using *tplmap*, we summarize two major limitations in its design. First, *tplmap* relies on the HTTP response to identify a successful template injection, which cannot identify second-order injections [19, 40]. Second, *tplmap* uses fixed payloads to test template injection points which are hard to satisfy the complicated constraints on the injected parameters [27].

Experiment-II: Manual Discovery. Due to the poor performance of *tplmap*, we resort to discovering some SSTI vulnerabilities manually. Inspired by the analysis of known SSTI vulnerabilities (see §5.6.1), we decided to test whether these applications provide some normal functionalities of template code injection. In all, we find template code injections in six applications, which are presented in Table 5. Among these applications, only Piwigo provides template selection as the normal functionality, while all other applications allow direct template injection. For Piwigo, we also discover a file upload vulnerability that can overwrite template files on the server side. Thus, we also gain template injection in Piwigo by combining the file upload vulnerability and the template selection functionality. With the synthesized RCE exploits by TEFuzz, we successfully achieve full exploitation on these applications.

Summary: We have two findings from the above experiments. ❶ Though SSTI vulnerabilities are severe, the automatic discovery of SSTI vulnerabilities still requires further research. ❷ By escaping the TE sandbox, template escape bugs lead to severe security consequences for real-world applications. We have demonstrated the full exploitability of the discovered template escape bugs in *six* popular PHP applications.

5.7 Internal Results of TEFuzz (RQ5)

TEFuzz has introduced several important designs to effectively detect and exploit template escape bugs. In this research question, we measure how do these designs help TEFuzz by reporting some internal results of TEFuzz.

Testcase Probing. TEFuzz creates a lot of new testcases from the seeds to discover some interesting testcases. As shown in Table 6, based on the collected 1,728 seeds, TEFuzz creates 64,491 new testcases, while 36,540 ones of them are found to contain *EP*s. Since many interesting testcases share the same *EP*, TEFuzz further identifies 4,484 (12.3%) unique interesting testcases with coverage-guided

Table 5: Manually Discovered SSTI Vulnerabilities in Real-world PHP Applications.

Application	Version	Stars	TE	RCE	Root Cause
CMSMS	2.2.16		Smarty	✓	Normal Functionality of Template Modification
lmxcms	1.41		Smarty	✓	Normal Functionality of Template Modification
Piwigo	13.0.0	2.2k	Smarty	✓	File Upload to Template Overwrite + Normal Functionality of Template Selection
MediaWiki	1.38.2	3.2k	Smarty	✓	Normal Functionality of Template Modification
TikiWiki CMS	21.7		Smarty	✓	Normal Functionality of Template Modification
Ejucms	SP4		ThinkPHP	✓	Normal Functionality of Template Modification

Table 6: Internal Results of TEFuzz in Generating Interesting Testcases and PoCs (RQ5).

TE	Seeds	Testcase Probing		PoC Generation	
		Created Testcases	Interesting Testcases (A/U)[1]	Created Testcases	PoCs (A/U)[1]
Smarty	523	16664	(11,663 / 700)	93,425	(23 / 3)
Twig	339	16,290	(8,172 / 850)	113,900	(0 / 0)
Dwoo	208	9,335	(5,600 / 557)	70,407	(68 / 46)
Latte	289	8,930	(5,635 / 721)	74,280	(263 / 63)
Mustache	17	563	(482 / 106)	10,385	(15 / 1)
Fenom	181	5,447	(3,116 / 343)	46018	(13 / 10)
ThinkPHP	171	7,262	(6,310 / 591)	53,086	(279 / 47)
All	1,728	64,491	(36,540 / 4,484)	546,893	(661 / 170)

[1] A(ll) represents the number of all the cases; U(nique) represents the number of unique cases.

Table 7: Internals Results of the Testcase Adaption in Different Modules (RQ5).

TE	Seed Collection			Testcase Probing			PoC Generation		
	TE Errors	Fixed Cases	Fix Rate	TE Errors	Fixed Cases	Fix Rate	TE Errors	Fixed Cases	Fix Rate
Smarty	268	42	15.67%	2840	1403	49.40%	14,593	13,453	92.19%
Twig	245	12	4.90%	2,622	1,349	51.45%	29,044	17,803	61.30%
Dwoo	52	11	21.15%	1583	816	51.55%	1,630	1,402	86.01%
Latte	237	15	6.33%	334	106	31.74%	158	103	65.19%
Mustache	0	0	-	171	130	76.02%	2893	2,802	96.85%
Fenom	78	16	20.51%	360	138	38.33%	5,830	4,011	68.80%
ThinkPHP	26	11	42.31%	115	11	9.57%	497	469	94.37%
All	906	107	11.81%	8,025	3,953	49.26%	54,645	40,043	73.28%

clustering. The results show that the probing technique effectively identifies a lot of interesting testcases and significantly reduces the testing workload.

PoC Generation. For each interesting testcase, TEFuzz inserts PHP syntax characters at its *EP*s to trigger template escape bugs. Table 6 presents the internal results. Based on the discovered 4,484 interesting testcases, TEFuzz creates 546,893 new testcases to discover bugs. In all, TEFuzz has discovered 661 PoCs that trigger bugs.

With the help of the PoC clustering technique, TEFuzz automatically clusters 170 unique PoCs. We manually analyzed these PoCs and confirmed 135 real bugs (as shown in Table 2). The manual bug diagnosis on 170 unique PoCs costs about 79 man-hours. The results show that the PoC clustering technique largely saves the manual analysis of the reported bugs. To understand the false negatives of the PoC clustering, we randomly selected 200 cases from the 663 PoCs for manual verification. We find that TEFuzz never identifies unique PoCs as the same (i.e., zero false-negative rate). It mainly owes to the conservative design in PoC clustering.

Table 8: Utility of the Testcase Adaption in Seed Collection, Bug Detection and Exploit Synthesis (RQ5).

TE	Seeds		PoCs		Exploits	
	w/	w/o	w/	w/o	w/	w/o
Smarty	523	479	3	1	3	1
Twig	339	327	0	0	0	0
Dwoo	208	197	38	34	2	1
Latte	289	274	49	40	24	20
Mustache	17	17	1	0	1	0
Fenom	181	165	10	8	10	8
ThinkPHP	171	160	34	28	15	12
All	1,728	1,619	135	111	55	42

Testcase Adaption. TEFuzz features testcase adaption to fix the testcases that meet testing errors. This technique is used in the *Seed Collection* module, the *Testcase Probing* module, and the *PoC Generation* module. Table 7 presents the internal results of the testcase adaption. In all, TEFuzz meets TE errors in 64,491 testcases and successfully fixes 44,103 cases (fix rate: 69.4%). Considering that template testcase fixing is difficult, our testcase adaption technique achieves quite good

performance. We also randomly selected 100 failed cases to analyze the causes. Our investigation shows two scenarios that make TEFuzz fail to fix an error testcase. First, some errors cannot be fixed. For example, new tags may be created during the mutation while the tested TE does not support these tags. Second, sometimes the error messages do not convey sufficient information for fixing.

Table 7 also shows the number of fixed cases in each module. We can observe that the fix rate in the *Seed Collection* module is quite low (i.e., 11.8%). The reason is that its error testcases are collected from the wild, which cannot be fixed. For example, some testcases require third-party plugins; some testcases use forbidden tags in the sandbox mode. Besides, the *PoC Generation* module reports the most testcase errors (i.e., 54,645 cases that occupy 86.0% of all the TE errors), while its fix rate is also the highest. The large number of error testcases is due to a large number of newly-generated testcases in this module. The reason for its high fix rate is that the TE errors at this stage are mostly caused by similar reasons and are easier to model and fix.

Table 8 presents the numbers of the collected seeds, the discovered PoCs and the synthesized exploits when testcase adaption is present (w/) or not (w/o). The results show that testcase adaption significantly helps to collect 6.7% more seeds, discover 21.6% more bugs, and synthesize 31.0% more exploits.

Exploit Synthesis. For every discovered PoC, TEFuzz successfully synthesizes an exploit for it. However, among the 135 synthesized exploits, 80 ones fail to run in the testing framework. By manually analyzing all the failed exploits, we find that they all report TE errors during the template parsing step. According to the error messages returned by the TE, we classify two causes for these failures.

- *Template Parsing Errors (58 cases).* In these cases, the payloads that are used to wrap the escape context in the PHP file make the TE fail to parse the template code. For example, a PoC for Dwoo contains a `capture` tag; however, when parsing the synthesized exploit, Dwoo fails to recognize it as a `capture` tag, due to the new characters introduced during the exploit synthesis.
- *Template Validation Errors (22 cases).* In this situation, the TE correctly parses the synthesized exploit; however, it raises TE errors when checking the format of the parsed elements. For example, a synthesized exploit for Latte has a `variable` tag. Though Latte correctly parses the exploit, it reports TE errors because the variable name of the template code contains unsupported characters of Latte.

5.8 Responsible Disclosure

We have responsibly disclosed all the discovered vulnerabilities in our experiments to the developers. We first use email to contact them and then try GitHub issues (if we cannot find the email address or do not receive a reply). Currently, the vulnerabilities in Smarty, Latte and Mustache have been patched, and the vulnerabilities in Fenom have been confirmed. For the SSTI vulnerabilities discovered in §5.6.2, we have contacted all the developers and MediaWiki has updated their used version of Smarty to the patched version.

6 Discussion

Problem Scope. This paper conducts an in-depth study on the template escape bug, an overlooked sandbox bypass vulnerability in modern TEs. Our study shows that template escape bugs are subtle, prevalent, and severe. Besides, there are other bugs that could bypass the sandbox, mainly caused by the incomplete implementation of the sandbox mode. Different from such bugs, template escape bugs occur in the template parsing and translation process, which are prone to happen but quite challenging to detect, making template escape bugs under-explored yet. Meanwhile, it is found that template escape bugs only affect generation-based TEs. Nevertheless, for the excellent performance, generation-based TEs are more welcome, especially on script-based web platforms, such as PHP, Node.js, and Python. Further, though our tool and evaluation only target PHP TEs, template escape bugs also affect TEs of other web platforms, which has been verified in our early study on Python and Node.js TEs. Besides, our approach can be adapted to TEs on other platforms.

Technical Limitations. Template escape bugs are logic flaws caused by improper sanitization when translating template code into PHP code. Hence, the detection and exploitation of template escape bugs are pretty challenging (see §3.2) and make them not systematically studied. As the first work to detect template escape bugs, TEFuzz still meets several limitations. First, TEFuzz requires some manual efforts to support a new TE, i.e., extracting its delimiters, creating a testcase driver, and compiling adaption rules to fix invalid testcases. According to our experience in the evaluation, we find such efforts quite affordable. Second, TEFuzz cannot detect bugs for those uncovered template tags/syntax in the seed corpus. In the future, TEFuzz can be augmented with some testcase generation techniques [41, 48]. Third, TEFuzz has not considered the control-/data-flow constraints in TEs during the generation of PoCs and exploits, which may miss some true bugs and exploits. This limitation can be addressed by incorporating constraint solving [16, 50].

Non-exploitable Exploits. In our evaluation, TEFuzz successfully synthesizes 135 exploits; however, 80 of them meet TE errors in the dynamic validation. As described in §5.7, there are two causes for these 80 failed exploits: *template parsing errors* and *template validation errors*. The underlying reason for generating non-exploitable exploits is that the current solution of finding the right payload to wrap the escape context only considers the PHP syntax while ignoring the template syntax. Therefore, these non-exploitable exploits

still have the potential to evolve into RCE exploits, e.g., by manipulating them to fit the syntax of template engine. However, this requires sophisticated hacking skills. Beyond the usage of exploitability assessment, PoCs facilitate bug diagnosis and patch development. In our experience of fixing template escape bugs, we find that PoCs conveyed much useful information to understand the root causes of bugs.

Suggestions. In the evaluation, we find that the number of template escape bugs varies significantly across TEs. In §5.3, we have compared the practice of different TEs. Based on our investigation of the discovered bugs, we have two further suggestions for TE developers, which may help avoid template escape bugs. First, we find that none of the studied TEs develops the template parser on top of standard parse generators (e.g., Bison, ANTLR). That is, the TE parsers are developed in an ad-hoc manner. Though some TEs perform strict validation on the template code syntax, the maintenance cost is high. Second, we suggest that TE developers should perform context-aware code sanitization when generating PHP code. For example, different characters would be sanitized when generating PHP code comments and function invocation statements. According to our experience, we only observe Twig following such practice. Further, since code sanitization is complex and prone to be incomplete, we suggest defining the sanitization logic as several unified functions according to the context. The advantage is that these unified sanitizing functions could be reused elsewhere. However, we do not encounter a TE that has such practice.

7 Related Work

Vulnerability Detection/Exploitation in Web Applications. For the popularity of web applications, detecting web application vulnerabilities has received much attention. Specifically, various techniques have been explored, such as static analysis [17, 24, 56], dynamic testing [22, 28, 51], hybrid analysis [16], and machine/deep learning [27, 60]. Most web application vulnerabilities are injection-based, such as cross-site scripting [27], SQL injection [60], file upload [28]In addition to injection-based vulnerabilities, web applications also have logic flaws [44]. Access control vulnerability is a classical logic flaw [32, 38]. Execution after redirect vulnerability allows the server-side execution to continue after the request redirection [21, 43]. Cross-site request forgery is an attack that tricks an authenticated user into executing unwanted actions on a web application [45]. ReDoS [49] is a new logic flaw that exploits algorithmic complexity on regular expressions [33, 34, 36]. To the best of our knowledge, this paper is the first to study template escape bugs, and our study shows that they are prevalent and severe.

Meanwhile, exploiting web vulnerabilities is becoming more and more sophisticated, which usually requires combining multiple vulnerabilities. For example, PHP object injection (POI) vulnerabilities are exploited together with POP chains [20, 42]; exploiting JavaScript prototype pollution vulnerabilities need to stitch with prototype gadgets [25, 30, 31]. Similarly, this paper exploits template escape bugs to augment the capability of template modification vulnerabilities.

Fuzzing-based Vulnerability Detection. Fuzzing has been widely used in vulnerability detection. In essence, fuzzing is mostly a random process, so it needs to balance vulnerability exploration and exploitation [59, 61]. Besides, runtime feedback is critical to the success of fuzzing, which significantly helps to increase the code coverage and avoid redundant testing [53, 55]. Fuzzing is based on generating new testcases. Thus, different seed mutation and selection strategies have also been explored [18, 35, 54]. Due to the effectiveness of fuzzing in generating crashes (i.e., finding PoCs), accurate crash/PoC clustering is very important to ease the bug triage [23, 52]. Inspired by existing works, TEFuzz has introduced several important designs to discover template escape bugs effectively, and the evaluation results have demonstrated their helpfulness.

8 Conclusion

This paper studies template escape bugs, an overlooked sandbox bypass vulnerability in template engines that could elevate SSTI attacks to remote code execution. To understand the prevalence and the severity of template escape bugs, this paper presents TEFuzz to automatically detect and exploit such bugs. TEFuzz proposes a tailored fuzzing-based approach to effectively detect template escape bugs, which does not need to learn the syntax of the template code, and gives PoCs and exploits for the discovered bugs. With the help of TEFuzz, we conduct an in-depth study with seven popular PHP template engines. In all, TEFuzz discovers 135 new template escape bugs and synthesizes real exploits for 55 bugs. Our study calls for attention to such subtle and severe bugs, uncovers the root causes of these bugs, and sheds light on the developing practice of template engines. The code and data are released to facilitate follow-up research [10].

Acknowledgement

We would like to thank our shepherd and the reviewers for their helpful comments and feedback. This work was supported in part by the National Natural Science Foundation of China (62172105, 62172104, 62102091, 62102093) and the National Key R&D Program of China under Grant 2021YFB3101200. Yuan Zhang was supported in part by the Shanghai Rising-Star Program 21QA1400700 and the Shanghai Pilot Program for Basic Research-Fudan University 21TQ1400100 (21TQ012). Min Yang is the corresponding author, and a faculty of Shanghai Institute of Intelligent Electronics & Systems, Shanghai Institute for Advanced Communication and Data Science, and Engineering Research Center of CyberSecurity Auditing and Monitoring, Ministry of Education, China.

References

[1] A Pentester's Guide to Server Side Template Injection (SSTI). https://www.cobalt.io/blog/a-pentesters-guide-to-server-side-template-injection-ssti.

[2] American Fuzzy Lop. http://lcamtuf.coredump.cx/afl/.

[3] crawlergo. https://github.com/Qianlitp/crawlergo.

[4] CVE-2017-16783. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-16783.

[5] CVE-2017-6070. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-6070.

[6] CVE-2022-22929. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22929.

[7] CVE-2022-22954. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-22954.

[8] CVE-2022-44978. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-44978.

[9] KernelMemorySanitizer. https://github.com/google/kmsan.

[10] TEFuzz Release. https://github.com/seclab-fudan/TEFuzz/.

[11] Undefined Behavior Sanitizer. https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.

[12] Understanding Template Injection Vulnerabilities. https://www.paloaltonetworks.com/blog/prisma-cloud/template-injection-vulnerabilities/.

[13] Web Template System. https://en.wikipedia.org/wiki/Web_template_system.

[14] Websites built with Latte. https://builtwith.nette.org/.

[15] What Are Template Engines? https://www.educative.io/edpresso/what-are-template-engines/.

[16] A. Alhuzali, R. Gjomemo, B. Eshete, and V. Venkatakrishnan. NAVEX: Precise and Scalable Exploit Generation for Dynamic Web Applications. In *USENIX Security'18*.

[17] M. Backes, K. Rieck, M. Skoruppa, B. Stock, and F. Yamaguchi. Efficient and Flexible Discovery of PHP Application Vulnerabilities. In *EuroS&P'17*.

[18] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. EnFuzz: Ensemble Fuzzing with Seed Synchronization among Diverse Fuzzers. In *USENIX Security'19*.

[19] J. Dahse and T. Holz. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security'14*.

[20] J. Dahse, N. Krein, and T. Holz. Code Reuse Attacks in PHP: Automated POP Chain Generation. In *CCS'14*.

[21] A. Doupé, B. Boe, C. Kruegel, and G. Vigna. Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities. In *CCS'11*.

[22] B. Eriksson, G. Pellegrino, and A. Sabelfeld. Black Widow: Blackbox Data-driven Web Scanning. In *S&P'21*.

[23] Z. Jiang, X. Jiang, A. Hazimeh, C. Tang, C. Zhang, and M. Payer. Igor: Crash Deduplication Through Root-Cause Clustering. In *CCS'21*.

[24] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a Static Analysis Tool for Detecting Web Application Vulnerabilities. In *S&P'06*.

[25] Z. Kang, S. Li, and Y. Cao. Probe the Proto: Measuring Client-Side Prototype Pollution Vulnerabilities of One Million Real-world Websites. In *NDSS'22*.

[26] J. Kettle. Server-Side Template Injection: RCE for the Modern WebApp. In *BlackHat US*, 2015.

[27] S. Lee, S. Wi, and S. Son. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *WWW'22*.

[28] T. Lee, S. Wi, S. Lee, and S. Son. FUSE: Finding File Upload Bugs via Penetration Testing. In *NDSS'20*.

[29] S. Lekies, B. Stock, and M. Johns. 25 Million Flows Later: Large-Scale Detection of DOM-Based XSS. In *CCS'13*.

[30] S. Li, M. Kang, J. Hou, and Y. Cao. Detecting Node.js prototype pollution vulnerabilities via object lookup analysis. In *ESEC/FSE'21*.

[31] S. Li, M. Kang, J. Hou, and Y. Cao. Mining Node.js Vulnerabilities via Object Dependence Graph and Query. In *USENIX Security'22*.

[32] X. Li, X. Si, and Y. Xue. Automated Black-Box Detection of Access Control Vulnerabilities in Web Applications. In *CODASPY'14*.

[33] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S.-C. Cheung. ReDoSHunter: A Combined Static and Dynamic Approach for Regular Expression DoS Detection. In *USENIX Security'21*.

[34] Y. Liu, M. Zhang, and W. Meng. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *S&P'21*.

[35] C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *USENIX Security'19*.

[36] R. McLaughlin, F. Pagani, N. Spahn, C. Kruegel, and G. Vigna. Regulator: Dynamic Analysis to Detect ReDoS. In *USENIX Security'22*.

[37] D. Molnar, X. C. Li, and D. A. Wagner. Dynamic Test Generation to Find Integer Bugs in X86 Binary Linux Programs. In *USENIX Security'09*.

[38] M. Monshizadeh, P. Naldurg, and V. Venkatakrishnan. MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications. In *CCS'14*.

[39] J. Offutt, J. Pan, and J. M. Voas. Procedures for Reducing the Size of Coverage-based Test Sets. In *Proceedings of the 12th International Conference on Testing Computer Software (TCS)*, 1995.

[40] O. Olivo, I. Dillig, and C. Lin. Detecting and Exploiting Second Order Denial-of-Service Vulnerabilities in Web Applications. In *CCS'15*.

[41] M. Olsthoorn, A. van Deursen, and A. Panichella. Generating Highly-structured Input Data by Combining Search-based Testing and Grammar-based Fuzzing. In *ASE'20*.

[42] S. Park, D. Kim, S. Jana, and S. Son. FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities. In *USENIX Security'22*.

[43] P. Payet, A. Doupé, C. Kruegel, and G. Vigna. EARs in the Wild: Large-scale Analysis of Execution after Redirect Vulnerabilities. In *SAC'13*.

[44] G. Pellegrino and D. Balzarotti. In *NDSS'14*.

[45] G. Pellegrino, M. Johns, S. Koch, M. Backes, and C. Rossow. Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs. In *CCS'17*.

[46] S. Rawat, V. Jain, L. C. Ashish Kumar, C. Giuffrida, and H. Bos. VUzzer: Application-aware Evolutionary Fuzzing. In *NDSS'17*.

[47] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC'12*.

[48] E. Soremekun, E. Pavese, N. Havrikov, L. Grunske, and A. Zeller. Probabilistic Grammar-based Test Generation. In *Software Engineering 2021*.

[49] C.-A. Staicu and M. Pradel. Freezing the Web: A Study of ReDoS Vulnerabilities in JavaScript-based Web Servers. In *USENIX Security'18*.

[50] M.-T. Trinh, D.-H. Chu, and J. Jaffar. S3: A Symbolic String Solver for Vulnerability Detection in Web Applications. In *CCS'14*.

[51] O. van Rooij, M. A. Charalambous, D. Kaizer, M. Papaevripides, and E. Athanasopoulos. webFuzz: Grey-Box Fuzzing for Web Applications. In *ESORICS'21*.

[52] R. van Tonder, J. Kotheimer, and C. Le Goues. Semantic Crash Bucketing. In *ASE'18*.

[53] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song. Be Sensitive and Collaborative: Analyzing Impact of Coverage Metrics in Greybox Fuzzing. In *RAID'19*.

[54] J. Wang, C. Song, and H. Yin. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *NDSS'21*.

[55] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *NDSS'20*.

[56] S. Wi, S. Woo, J. J. Whang, and S. Son. HiddenCPG: Large-Scale Vulnerable Clone Detection Using Subgraph Isomorphism of Code Property Graphs. In *WWW'22*.

[57] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley. Scheduling Black-Box Mutational Fuzzing. In *CCS'13*.

[58] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang. ProFuzzer: On-the-fly Input Type Probing for Better Zero-Day Vulnerability Discovery. In *S&P'19*.

[59] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *USENIX Security'20*.

[60] K. Zhang. A Machine Learning Based Approach to Identify SQL Injection Vulnerabilities. In *ASE'19*.

[61] L. Zhang, K. Lian, H. Xiao, Z. Zhang, P. Liu, Y. Zhang, M. Yang, and H. Duan. Exploit The Last Straw that Breaks Android System. In *S&P'22*.

[62] S. Zhu. crawlergo: A Powerful Browser Crawler for Web Vulnerability Scanners. In *BlackHat Europe*, 2021.