FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities

Sunnyeo Park* *KAIST*

Daejun Kim* *KAIST*

Suman Jana *Columbia University*

Sooel Son *KAIST*

Abstract

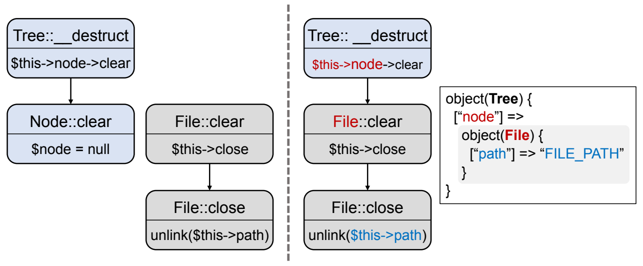
A PHP object injection (POI) vulnerability is a security-critical bug that allows the remote code execution of class methods existing in a vulnerable PHP application. Exploiting this vulnerability often requires sophisticated property-oriented programming to shape an injection object. Existing off-the-shelf tools focus only on identifying potential POI vulnerabilities without confirming the presence of any exploit objects. To this end, we propose FUGIO, the first automatic exploit generation (AEG) tool for POI vulnerabilities. FUGIO conducts coarse-grained static and dynamic program analyses to generate a list of gadget chains that serve as blueprints for exploit objects. FUGIO then runs fuzzing campaigns using these identified chains and produces exploit objects. FUGIO generated 68 exploit objects from 30 applications containing known POI vulnerabilities with zero false positives. FUGIO also found two previously unreported POI vulnerabilities with five exploits, demonstrating its efficacy in generating functional exploits.

1 Introduction

A *PHP object injection* (POI) vulnerability is a security-critical PHP application bug [46] that enables diverse attacks, including cross-site scripting, SQL injection, and arbitrary file deletion. PHP supports the functionality of deserializing a string into a PHP object to facilitate the management of run-time objects. When exploiting a POI vulnerability, an adversary targets this functionality; the adversary passes a forged string into a deserialization function call, thus injecting an arbitrary PHP object into the target application scope.

A unique aspect of this attack is that the adversary should alter the properties and structures of the PHP object to invoke a series of user-defined functions or class methods, thereby allowing various types of web attacks. The technique used in composing this *exploit object* is called *property-oriented programming* (POP).

*Both authors contributed equally to the paper



```
object(Tree) {
  ["node"] =>
    object(File) {
      ["path"] => "FILE_PATH"
    }
}
```

Figure 1: A high-level overview of POP

Figure 1 depicts a high-level overview in which the adversary performs POP for arbitrary file deletion. The left side of the figure shows two legitimate call stacks that a target application implements. The right side shows a new stack trace that the adversary composes using POP, which the target application does not implement. After injecting the class object on the right side, the target application executes this new stack trace, thus deleting the file that the adversary chooses. The adversary connects two stack traces by injecting an object of which the node property has a `File` class object and feeds a file path by changing the `path` property. This new stack trace is called a *POP chain*, which invokes a series of user-defined existing class methods or functions.

The presence of a deserialization invocation with unsanitized user input merely poses a potential threat, whereas the existence of an actual exploit object poses a critical security threat. However, existing web scanning tools [32, 50, 58] only report potential POI vulnerabilities that merely deserialize user input. Unfortunately, generating exploit objects via POP requires a significant amount of human effort and expertise. However, there is no practical alternative for confirming the exploitability of such potential POI vulnerabilities.

Contributions. We propose FUGIO, the first automatic exploit generation (AEG) tool designed to find POI vulnerabilities and generate exploits for the identified vulnerabilities.

AEG for POI vulnerabilities requires addressing two challenges: 1) it is necessary to identify available POP chains that consist of user-defined methods and functions called *gadgets* while considering the availability of these gadgets and their caller-callee relationships; and 2) for each identified chain, it should shape an injection object by properly setting not only the object hierarchy but also its multiple properties, thus following the execution flow that this POP chain specifies.

We choose to tackle AEG for POI bugs with feedback-driven targeted fuzzing aided by static and dynamic analyses. Specifically, FUGIO addresses both of the following challenges: 1) FUGIO identifies all POP chains by analyzing dynamically generated classes as well as statically defined classes and prunes unavailable gadgets when triggering a target POI vulnerability; and 2) FUGIO generates exploit objects that follow the execution of identified chains by conducting feedback-driven fuzzing on a debloated PHP application.

Given the source code of a target PHP application, FUGIO starts by collecting all class and function information via static analysis to identify all available gadgets. It also collects dynamically generated classes and functions when the POI vulnerability is triggered via dynamic analysis, thus allowing FUGIO to consider a comprehensive POP chain set.

Based on both of the analyses, FUGIO identifies feasible POP chains and generates a *program under testing* (PUT) that embodies all gadgets. When composing the PUT, it mimics the execution environment in which the vulnerability is triggered. 模仿 FUGIO then conducts a feedback-driven fuzzing campaign on the PUT for each chain, thus generating exploit objects. The fuzzing process harnesses execution feedback when prioritizing promising inputs that reach gadgets deeper in the given chain and mutating property values. Once FUGIO finds a potential exploit, it inserts an attack payload into the properties of the exploit object to check whether the injected payload appears at the sink function, thus confirming the exploitation of the generated payload.

We evaluated FUGIO on 30 PHP applications, each of which has at least one known POI vulnerability. From these applications, FUGIO reported 68 exploitable chains and their actual exploits with zero false positives. We also applied FUGIO to the latest versions of two PHP applications: WordPress with WooCommerce [3] and Concrete5. FUGIO reported two previously unreported POI vulnerabilities with concrete exploits, thus demonstrating its efficacy.

In summary, we propose a new AEG tool for POI vulnerabilities that applies a series of static and dynamic program analyses and feedback-driven fuzzing. We demonstrate that the proposed technique is effective for generating exploits with zero false positives, which has notoriously demanded labor-intensive engineering efforts.

2 Background

2.1 PHP Object Injection

A PHP object injection (POI) vulnerability refers to a security-critical bug in a PHP application that entails remote code execution [46]. An adversary injects an input string that exploits this vulnerability, and the vulnerable application creates a PHP object when deserializing this injected string. The adversary programs this PHP object via POP, enabling it to invoke a series of class methods or functions that result in file deletion/creation, cross-site scripting, remote code execution, and other malicious behaviors.

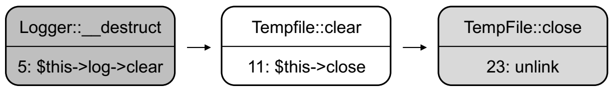
```php
<?php
class Logger {
  public function __destruct() { // Magic method
    if ($this->logtype === "TEMPORARY") {
      $this->log->clear();
    } else {
      $this->log->save();
} } }
class Stream {
  public function clear() {
    $this->close();
  }
  public function close() {
    $this->handle->close();
} }
class TempFile extends Stream {
  public function save() {
    $tmpfile = tempnam("/tmp", "XYZ_");
    $data = file_get_contents($this->filename);
    file_put_contents($tmpfile, $data); // Sink
  }
  public function close() {
    unlink($this->filename); // Sink
} }
$data = unserialize($_COOKIE['data']); // POI bug
```

Listing 1: Exploitation of a POI vulnerability

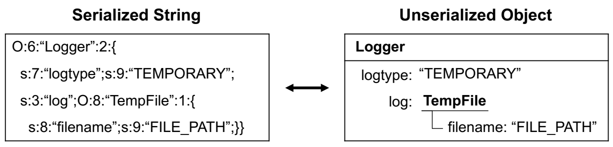
Unserialization. PHP supports two built-in functions that developers often use to encode and decode object-type data: `serialize()` and `unserialize()`. These methods are designed to serialize a given object into a string and to convert a given string into an object, respectively. Unserialization also occurs when a file operation is performed on a PHP Archive (PHAR) file, such as `file_exists` and `is_file`, because this file type stores its meta-data in serialized form. Therefore, an adversary can inject an arbitrary object at run-time by feeding an input string into an `unserialize` invocation or by triggering file operations on an attacker's uploaded PHAR file. For the latter case, we assume that a forged PHAR file has already been uploaded to a target web server by exploiting an unrestricted file upload (UFU) vulnerability [40]. The adversary thus attempts to invoke PHP file operations on this uploaded PHAR file, which unserializes its meta-data.

Property-oriented programming. When exploiting a POI vulnerability, an attacker crafts an injection object by carefully choosing its property values to invoke a chain of existing class methods or functions. This chain reflects a call stack that starts from the entry method and ends with a method that invokes a security-sensitive function with attack payloads. This chain is called a *POP chain*, and each method of this POP chain is called a *gadget*.

Note that the injection of an arbitrary object does not always lead to code execution; for execution to occur, the PHP interpreter that runs a vulnerable application should invoke



(a) A POP chain triggering `unlink`



(b) An exploit object for 2a

Figure 2: An example of POP for the POI in Listing 1

the entry gadget in a POP chain. To achieve this, the attacker uses a *magic method* defined in the vulnerable application. Various types of magic methods are automatically invoked when certain conditions are met at run-time. For instance, a `__destruct` method is called when deleting an object in the class definition that implements this magic method.

The attacker is also required to adjust the properties of the injected object to invoke the next gadget within the previous gadget's body. The next gadget should be 1) a member method of the class containing the previous gadget, 2) a member method of a different class whose name is used at the invocation statement in the previous gadget, or 3) a user-defined function with no owner class.

Listing 1 shows a PHP snippet with the POI vulnerability at Line 25. The attacker is able to inject an arbitrary object via the "data" cookie. Figure 2a shows a POP chain devised by the attacker to exploit this vulnerability. This chain starts from the `__destruct` of a `Logger` object and ends with the `close` member method of a `TempFile` object in which a security-critical file deletion occurs at Line 23. Based on the chain, the attacker implements the serialized string on the left side of Figure 2b; this string is deserialized into a `Logger` object containing the `logtype` property, the value of the which is set to "TEMPORARY", and the `log` property, the value of which is set to a `TempFile` object. This object has the `filename` property, the value of which is set to a file path to delete, as shown on the right side of Figure 2b.

When this object is destroyed in runtime, its `__destruct` method is automatically invoked. Since the `logtype` property value is "TEMPORARY", `__destruct` invokes `clear` of `TempFile`, which sequentially calls `close` of `TempFile` and `unlink` with the target file name embodied in `filename`. Thus, the attacker is able to delete an arbitrary file by injecting the aforementioned input string. Note that the attacker assigns a `TempFile` object in the `log` property of `Logger` to connect the callsite at Line 5 to the second gadget, the `clear` member method of `TempFile`. This control flow is not intended by the application developers but is introduced by the attacker. The attacker does not introduce any new code but instead reuses existing gadgets by reshaping the hierarchy of objects and setting the appropriate property values.

The presented attack is analogous to code reuse attack techniques, such as *return-to-libc* [44], *return-oriented programming* [52], and *jump-oriented programming* [9]. However, it is different in that the basic block for executing code is a class member method in *POP*.

3 Motivation and Challenges

We propose an AEG approach for finding a POI vulnerability and generating injection objects that exploit this vulnerability. Existing off-the-shelf penetration testing tools, such as Burp [50] and Acunetix [32], focus only on identifying built-in callsites that deserialize user input, reporting potential POI vulnerabilities. The presence of their exploit objects is able to eliminate possible false positives; however, this task requires POP, which demands significant manual effort and expertise.

Dahse *et al.* [13] addressed this limitation by conducting a static analysis that reports promising POP chains. However, to eliminate false positives, this static approach still requires addressing the reachability analysis problem of finding the appropriate input to exploit one of the promising POP chains. Furthermore, as the number of promising chains increases, it becomes an arduous task to vet each chain and generate exploits. For instance, Contao CMS has at least 26,180 chains, making it improbable to manually check these chains and generate working exploits for five exploitable chains.

Performing AEG for POI vulnerabilities must achieve the following objectives: 1) find a POI vulnerability; 2) identify POP chains of available gadgets when triggering the identified vulnerability; and 3) generate input objects for exploitable POP chains among those identified, thus reporting the exploitability of the vulnerability.

Considering that the aforementioned static and pen-testing tools [13, 32, 50, 58] can already find potential POI vulnerabilities that pass user input into deserialization function calls, we do not emphasize the contribution of detecting potential POI bugs. Rather, our core contribution lies in generating exploit objects, which is required to address the second and third objectives. We explain the technical challenges of each objective below.

Identifying POP chains. An AEG tool should identify all POP chains that take into account gadgets from all loadable classes at the location where user inputs are deserialized.

(*C1-1*) *The dynamic nature of PHP makes it difficult to identify loadable classes and their gadgets*. The autoload feature enables the loading of any existing classes via invoking developer-specified load callbacks. Also, many PHP CMS applications rely heavily on dynamically generated PHP classes. Thus, considering only statically defined classes results in not considering available gadgets for POP and produces false negatives.

(*C1-2*) *A naive algorithm for connecting loadable gadgets produces a prohibitive number of POP chains to vet when assessing exploitability.* For instance, in Listing 1, `__destruct`

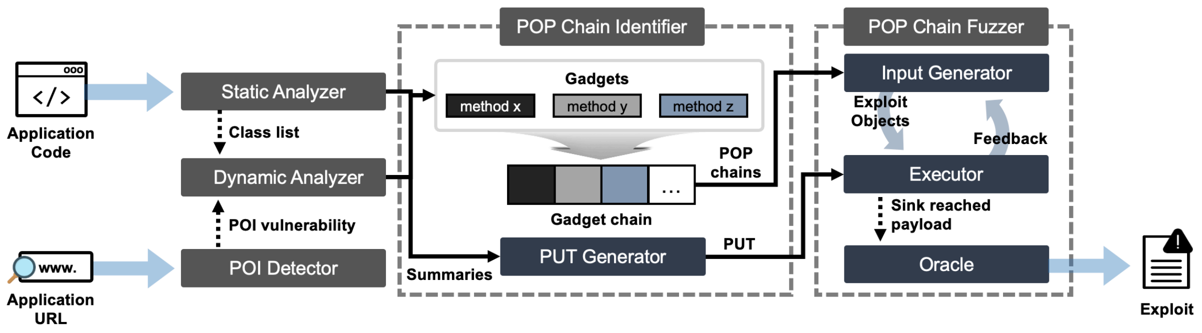


Figure 3: FUGIO architecture: A workflow overview of AEG for POI vulnerabilities

has two callsites that invoke `clear` and `save`. From this entry gadget of `__destruct`, the attacker can connect to different gadgets with the name of either `clear` or `save` by manipulating `$this->log`. Therefore, as the number of invocations within gadgets increases, the number of possible candidates to connect from a given gadget increases. This especially becomes problematic when enumerating long chains because the number of POP chains increases exponentially as length increases.

Generating exploits. An AEG tool should generate an exploit object with multiple property values that enable the exploit execution of an identified chain without disruption. This poses the difficult problem of generating an appropriate input that enables an application to reach a target statement.

Consider a gadget containing multiple conditions in its body. When the invocation statement of the next gadget involves passing these conditions, the tool is required to generate the appropriate property values to pass these. For instance, in Listing 1, if the next gadget of `Logger::__destruct` is `TempFile::clear`, the `logtype` of the object to inject should be "TEMPORARY", and the `log` should be a `TempFile` class object. Furthermore, the tool should also provide an attack payload to be fed into an actual parameter of a security-sensitive sink in the last gadget by injecting property values.

We choose feedback-driven fuzzing for generating exploits. Note that symbolic execution is undoubtedly applicable to systematically generating exploit objects [4, 11, 67]. However, this approach requires modeling the semantics of PHP built-in functions in terms of our symbols, which demands a heavy engineering effort; we observed that 30 applications in our benchmark used 460 built-in PHP functions in about 300 thousand of the identified gadgets in our evaluation. Furthermore, as the PHP interpreter evolves, more built-in functions will need to be supported for symbolic execution. Therefore, we choose a more general approach of fuzzing, which entails the risk of false negatives.

Fuzzing. Fuzzing a target PHP application invites distinctive technical challenges.

(C2-1) *It is difficult to establish a high throughput when conducting stateless fuzzing on a large PHP application.* When the size of a target PHP application is large, executing the application with each generated input is a slow process, which impedes fuzzing performance; 30 PHP applications in our benchmark required 0.6 sec, on average, to execute one object input, which is significantly slower than state-of-the-art fuzzers [10]. This is because the target application also executes modules that are not related to POI vulnerabilities. Furthermore, these modules can produce side effects that change an internal program state or even render the target application unable to run due to a large number of fuzzing attempts. Lastly, it is time-consuming to restore the application to its initial state after each fuzzing attempt.

(C2-2) *It is not straightforward to devise an exploit object with multiple property values via fuzzing.* A vast volume of existing fuzzing literature models fuzzing inputs as byte streams. However, generating an exploit object for POI vulnerabilities demands performing property-oriented programming, which requires altering multiple properties and shaping the object hierarchy to enable the execution of a POP chain. How can we identify the required properties for an exploit object? What properties should be selected for mutation? How should we mutate and generate the selected property values? These questions should be addressed to generate exploits via fuzz testing.

4 Overview

We propose FUGIO, a system that detects POI vulnerabilities in a target PHP application and confirms the exploitability of the detected vulnerabilities by generating exploit objects.

To address the two aforementioned challenges, FUGIO conducts both static and dynamic program analyses to consolidate all statically declared and dynamically generated gadgets *(C1-1)*. FUGIO then conducts a coarse-grained inter-procedural taint analysis (with the objective of pruning unnecessary POP chains) and performs a depth-bounded breadth-first search that enumerates all promising chains *(C1-2)*.

FUGIO generates exploits by conducting feedback-driven fuzzing, thus addressing the second challenge. To achieve a high throughput of fuzz testing, FUGIO synthesizes a program under testing (PUT) that simulates the execution environment in which a POI vulnerability is triggered *(C2-1)*. FUGIO leverages branch coverage, run-time reference errors, and hints that appear in conditional expressions during fuzzing to devise more promising input objects in terms of reaching deeper gadgets in the POP chain being tested *(C2-2)*.

FUGIO consists of five components: POI detector, static analyzer, dynamic analyzer, POP chain identifier, and POP chain fuzzer. These components work together to find POI vulnerabilities and to generate their exploit objects. Figure 3 depicts the workflow of FUGIO. It takes a target PHP application source code and URL as input. The POI detector starts to crawl the websites to detect POI vulnerabilities and passes the detected vulnerability to the dynamic analyzer. The static and dynamic analyzers collect data for identifying POP chains from the given source code and the execution environment, respectively when the detected vulnerability is triggered. Using these analyses, the POP chain identifier computes promising POP chains and generates a PUT. For each identified chain, the fuzzer performs fuzz testing on this PUT until it finds concrete exploits.

5 Design

5.1 POI Detector

The POI detector detects potential POI vulnerabilities in a target PHP application through dynamic testing. Specifically, it dynamically detects injection points that take in a predefined input string and convert it into a PHP object.

Given a target PHP application with its URL, the detector starts by crawling the application and computing a list of URLs to visit. When visiting each webpage in this list, the detector extracts `<a>` and `<form>` tags. From each `<a>` tag, the detector builds a GET request template to its destination URL. From each `<form>` tag, the detector extracts its action, method, and input parameters, each of which has a key and the default value. The detector then assembles these components and generates a request template with input parameters.

Accordingly, from each request template, the detector generates a list of testing requests with our input string. For each GET, POST, and COOKIE parameter in this request template, the detector injects a predefined serialized string that represents our testing PHP object, thus generating a set of testing requests; in each of these requests, one input parameter holds our predefined string. The detector then sends each testing request and observes whether a PHP built-in callsite that deserializes a user input is invoked with our testing object. For this, FUGIO hooks 26 predefined PHP built-in functions that involve the deserialization of user inputs, such as `unserialize`, `is_file`, and `file_exists`, using *runkit* [14] or *uopz* [34]. In short, when one of these built-in functions is invoked with an actual parameter that holds our testing object, the detector reports this invocation as a potential POI vulnerability and passes this callsite information to the dynamic analyzer.

5.2 Static Analyzer

The static analyzer computes *static summaries* for user-defined classes and functions that are later used for generating POP chains and a PUT by the POP chain identifier (§5.4).

The analyzer takes in the source code of the target application and outputs a static summary for each file. This static summary contains *function summaries* and *class summaries* that hold the information of functions and classes defined in this file. It starts by parsing each PHP file into a set of abstract syntax trees (ASTs) and then analyzes all definitions of functions, classes, interfaces, and traits. For each defined class, the static analyzer extracts information, including a name, namespace, its parent class, implemented interfaces, used traits, and the definition itself. It then stores this information in the class summary of the PHP file defining the class. This summary also includes the name and visibility of each defined property and member method.

When parsing a function or member method definition, the static analyzer extracts its function prototype and computes a function summary that contains caller-callee relationships. For each method invocation (e.g., `$receiver->method()`) within the function or member method definition, the analyzer extracts a target method name and its receiver class candidates. When the invocation explicitly uses `$this`, the static analyzer precisely infers the receiver as the owner of the target method or this owner's parent class. Otherwise, it assigns a set of classes that include member methods with the target method name to the receiver candidates, computing a conservative set of all possible classes. However, when there are intra-procedural data flows from the assignment statement of a newly instantiated class involving the `new` keyword to this receiver (e.g., `$receiver =` *new* `ClassA`), the analyzer uses this class as the receiver. This caller-callee information in each file summary is later used when connecting gadgets to build POP chains.

For each member method and function, the static analyzer computes flow-insensitive intra-procedural data flows from its formal parameters and its owner properties to the arguments of each invocation callsite in the body. The function summary contains these data flows, which are later used to prune non-exploitable intermediate gadgets in which the attack is unable to change the actual arguments of subsequent gadgets by adjusting the property values of an injection object.

5.3 Dynamic Analyzer

The dynamic analyzer computes *function summaries* and *class summaries* for dynamically generated classes and functions that are not statically defined in the target application source code. These summaries are also later consumed by the POP chain identifier for generating chains and the PUT.

Given a POI vulnerability, the dynamic analyzer collects additional information that the static analyzer is unable to obtain. For this, FUGIO creates a PHP file that installs hooks to PHP built-in functions that internally deserialize user input, such as `unserialize`, `file_exists`, and `fopen`, using the *runkit* [14] or *uopz* [34] extensions. We inject this file into the target web application using the `.htaccess` file.

When the given POI vulnerability is triggered, the installed hook attempts to load all classes obtained via the static analyzer. Note that PHP supports the autoload feature, which invokes developer-specified callbacks when accessing unloaded classes. Because many PHP applications use this autoload feature, it is important to know which classes can be loaded dynamically. We use `class_exists` to check whether a given class is loaded. Otherwise, this function automatically tries to load the given class using the corresponding autoload. By invoking `class_exists` for all classes, we obtain a list of loadable classes.

The dynamic analyzer subsequently examines loaded functions, classes, interfaces, and traits when their bodies are not statically defined. The analysis process is the same as the analyses performed in the static analyzer. However, the dynamic analyzer examines classes and functions that are dynamically defined, which do not exist in the source code. Note that Dahse *et al.* [13] did not consider these dynamically generated functions or classes.

Finally, the dynamic analyzer stores environmental variables (e.g., `$_ENV` and `$_SERVER`) and global variables (e.g., `$_GLOBAL`) when these hooking methods are called. In the next step, these variables are used to generate a PUT that mimics the execution environment in which the given POI vulnerability is triggered.

5.4 POP Chain Identifier

Based on information from the static and dynamic analyzers, the POP chain identifier emits a list of available POP chains and a PUT to use in performing fuzz testing.

5.4.1 POP Chain Identification

A POP chain is a sequence of gadgets that reflects a stack trace from a magic method to a sensitive sink when the exploitation of a target POI vulnerability occurs. Based on the type of this sensitive sink, the adversary is able to conduct different kinds of attacks. In this paper, we specified a total of 26 sensitive sinks causing file creation/modification/deletion, shell command injection, and remote code execution.

For each invocation of sensitive sinks, FUGIO checks whether the attacker can change the actual arguments of the invocation using the function summary of the function or method (m) that embodies the invocation (§5.2). It checks whether there is an intra-procedural data flow from any formal parameters of m or the properties of the owner class of m. If so, FUGIO takes this sink function into account when generating POP chains. Otherwise, FUGIO excludes this sink function because there is no way for the attacker to directly change such arguments via injecting objects. For instance, from Listing 1, FUGIO finds two sensitive sinks, `file_put_contents` (in Line 20) and `unlink` (in Line 23), by checking the intra-procedural data flow analysis of the `save` and `close` methods, respectively. FUGIO then checks for the existence of data flows from the `filename` property to the second argument of `file_put_contents` and the first argument of `unlink`.

The chain identifier is required to generate POP chains, each of which consists of gadgets along the path from a magic method to each target sensitive sink. One naive solution for this is to conduct a depth-first search from each magic method to a target sensitive sink, generating all feasible chains. However, traversing the downstream of a callee from an entry magic method may not encounter any sensitive sinks, thus wasting computation resources. Also, when there exists a cycle in call chains, the algorithm does not terminate.

Instead, we designed an algorithm that builds a depth-bounded call tree in a breadth-first manner as shown in Figure 4. Specifically, for each invocation of target sinks, the chain identifier of FUGIO computes a call tree, the root of which is the method embodying the invocation. Then, for each leaf node that indicates a function or method (m), it iteratively attaches new nodes, each child node of which corresponds to a potential caller of m. The chain identifier builds this tree until its height grows up to a given height, which is a parameter that auditors specify. We chose seven for this parameter in our evaluation (§7.4). By means of this height-bounded tree search, FUGIO is able to enumerate POP chains, even when there are cycles in calling gadgets.

手工设置7层，环路也无所谓了

Attaching leaf nodes. When attaching potential callers of the gadget (m) to each leaf, the chain identifier considers not only authentic callers that the developers intended for m but also forged callers that the adversary enforces for m via manipulating object properties. Specifically, the chain identifier collects all function summaries that have invocation statements of which 1) the target callee name is the same as m and 2) the number of actual parameters is the same as the number of formal parameters of m. This collected set of function summaries becomes the set of potential callers of m to attach. From the potential callers, the chain identifier further removes callers of which the invocation statements have statically deterministic receivers that do not indicate the owner class of m. Remind that via the static analysis, FUGIO already computed a conservative receiver set for each invocation (§5.2).

Considering that the chain identifier leverages static information when attaching potential callers of m, it misses call edges when potential callers are statically undecidable; for a reflective callsite, such as `$receiver->$method`, the chain identifier is unable to determine its possible callees, thus missing chains that contains this call edge. We discuss false negatives due to these reflective callsites in §7.3.2.

Generating chains. After building the call tree for each sink, the chain identifier finds the leaf nodes that represent magic methods. For each identified leaf node, it computes a path to the root node, thus emitting a chain of tree nodes, each of which corresponds to a POP gadget.

Before passing the identified POP chain to the next step, the chain identifier performs an inter-procedural data flow analysis to prune the chains that do not have any data flows from gadget properties to actual arguments of the sensitive sink, rendering it infeasible for the attacker to change these actual arguments. We leverage the function summary of each gadget that lies in this POP chain and compute inter-procedural data flows within the POP chain.

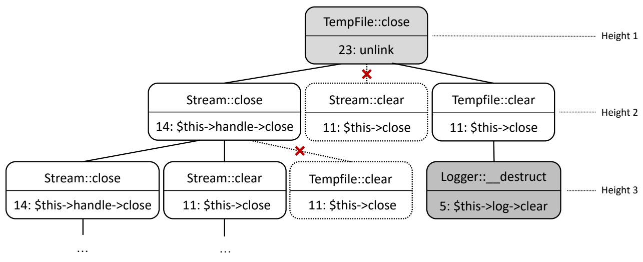


Figure 4: A call tree from `unlink` in Listing 1

5.4.2 PUT Generation

FUGIO synthesizes a PUT in PHP that is relatively smaller than its original application but contains all the gadgets necessary to generate exploits. FUGIO first extracts all class and function definitions from the summaries that the analyzers compute. To avoid possible conflicts of class and function names, FUGIO emits one file for each definition. We call this generated file a *definition file*.

FUGIO then generates the head and body parts of the PUT. In the head part, FUGIO sets all the environment and global variables provided by the dynamic analyzer. FUGIO then writes the body part to include all definition files that contain all the gadgets that are loaded and loadable when exploiting a target POI vulnerability. The body part ends with the `unserialize` invocation, which takes in a serialized input string that the fuzzer later provides. This PUT takes in this input string from a shell command invoking the PHP interpreter running the PUT. Therefore, the fuzzer invokes `unserialize` using a generated input in this PUT, which simulates the execution environment when exploiting the POI vulnerability.

Instrumentation for feedback. Before conducting feedback-driven fuzzing, FUGIO instruments a given PUT to obtain three kinds of feedback when conducting fuzz testing of each object input: 1) the execution trace that contains executed conditional expressions and invoked methods, 2) the number of executed gadgets in the POP chain, and 3) hints for property values from conditional expressions.

To obtain the first and second types of feedback, we design FUGIO to insert code that reports their execution at every function entry, before and after every conditional expression,

```
Algorithm 1: Fuzzing
  Input  : An instrumented PUT (put)
           A POP chain (pop_chain)
  Output : Payloads
1  seed_pool ←[]
2  init_seed ← GenerateInitialSeed(pop_chain)
3  AddSeed(seed_pool, init_seed)
4  while True do
5      seed ← SelectSeed(seed_pool)
6      input ← MutateSeed(seed)
7      execution ← ExecutePUT(put, input)
8      result ← AnalyzeExecution(execution)
9      if NewPath(result) then
10         AddSeed(seed_pool, input)
11         if SinkReached(result) then
12             Report(input, "ProbablyExploitable")
13             if Oracle(input) == success then
14                 Report(input, "Exploitable")
15                 break
16         else
17             if ExecutedGadgets(input) >
                 ExecutedGadgets(seed) then
18                 AddSeed(seed_pool,
                     GenerateNextGadgetSeed(input))
19             hinted_properties ← AnalyzeConditions(result)
20             foreach prop ∈ hinted_properties do
21                 AddSeed(seed_pool,
                     GenerateHintedSeed(input, prop))
22     else if reference error occurs then
23         missing_property ← AnalyzeError(execution)
24         AddSeed(seed_pool,
             AddProperty(input, missing_property))
```

and at every invocation statement of user-defined functions. Specifically, each instrumented code emits the hash value of its line and file, thus leaving an execution remark.

This step also inserts code that reports hints for property values, which the fuzzer leverages to generate precise property values. In Listing 1, `$this->logtype` needs to be set to "TEMPORARY" to invoke the `clear` (Line 5), but it is improbable to randomly generate this specific string. For this, the instrumented code reports constants and type-checking built-in functions, such as `is_string()` and `is_int()`, which appear in executed conditional statements. These reported constants and built-in functions are later used to generate property values. We call this type of property generation *property hinting*.

5.5 POP Chain Fuzzer

Given a PUT and a POP chain, the fuzzer conducts feedback-driven fuzzing on the PUT, generating an exploit object.

Algorithm 1 describes the overall process of fuzzing. Given a POP chain and a PUT, the fuzzer initiates a fuzzing campaign during a given time period. The fuzzing process starts by preparing an initial seed pool (Lines 1–3). It then repeats the following procedures until finding an injection object that executes the sensitive sink in the POP chain with an attack payload. The fuzzer picks a seed based on its previous execution feedback (Line 5) and mutates this seed, thus generating a new injection object to test (Line 6). The fuzzer then executes the instrumented PUT with the mutated seed and analyzes the execution feedback (Lines 7–8). When this mutated seed contributes to increasing branch coverage, the fuzzer adds this seed to the seed pool (Lines 9–10). Furthermore, if this mutated seed executes more gadgets than the existing seed, the fuzzer also derives new seeds that target one step deeper in the POP chain (Lines 17–18). It also computes new seeds by assigning property values in the current mutated seed by leveraging constants and inferring property types based on the usages of properties that appear in the executed conditional statements (Lines 19–21). The remaining section details each step in the fuzzing process.

Seed optimization. We prioritize a seed input that reaches a deeper gadget in a given chain or the security-sensitive sink in the last gadget of the chain when selecting a seed to mutate.

The fuzzer starts by preparing an empty seed pool (Line 1) and generates an initial seed input. This initial seed input is an object of which the class holds the first gadget of the given POP chain. Then, this initial seed is added to the seed pool (Lines 2–3). The fuzzer then picks a seed from the seed pool (Line 5). For each seed, the fuzzer stores its execution results: 1) the number of selections of the seed, 2) the hash of its property tree, and 3) the maximum depth of executed gadgets in the POP chain. Based on this information, the fuzzer prioritizes a less frequently executed seed that has a deeper executed gadget.

More formally, we define $diff$ as the difference between the length of the POP chain and the maximum depth of executed gadgets in the POP chain. Then, we use Equation 1 to assign the probability for a target gadget depth; the greater depth the executed gadget has, the greater chance of being selected the target depth has.

$$score_i = \frac{1}{1 + e^{\frac{5 \times diff_i}{\max(cur\_depth)}}}, \quad P_i = \frac{score_i}{\Sigma score_i} \tag{1}$$

After selecting a target gadget depth, the fuzzer then selects one seed among those in the seed pool that have reached this target depth in their execution.

When the seed pool includes seeds that reach the sink, FUGIO splits the probability of 1.0 into 0.9 and 0.1. It then uniformly distributes the probability of 0.9 across seeds that have reached the sink. The probability of 0.1 is also uniformly distributed across the remaining seeds in the pool. When there is no seed reaching the sink, the fuzzer assigns the same probability to each seed that shares the target gadget depth. It also decreases the probability of each seed as the number of this seed selection increases, thus increasing the chance of being selected for less-selected seeds.

Input generation. Generating an injection object requires 1) designing the structural hierarchy of multiple classes that reflects a given POP chain and 2) assigning proper values to the properties of these multiple classes, which facilitates reaching the sensitive sink with an attack payload. We leveraged a tree data structure called a *property tree* when generating and mutating this injection object. Its root node represents a class object of which the class holds the magic method, the entry gadget of a given POP chain. Each node of its descendants represents a class object property; it contains the property name, its visibility, type, and value. When a property type is a class object, this node holds class candidates that the property may have. In addition, this node becomes the parent node of a subtree that indicates another class object having another gadget in the POP chain.

```php
<?php
class Logger {
  public $logtype;
  public $log;
  function setProp($name, $value) {
    $parent_class = get_parent_class();
    if ($parent_class &&
        property_exists($parent_class, $name)) {
      parent::setProp($name, $value);
    }
    $this->$name = $value;
  }
  function getProp($name) {
    return $this->$name;
} }
class Stream {
  ...
}
class TempFile extends Stream {
  public $filename;
  ...
}
$input = new Logger;
$input->setProp('logtype', "TEMPORARY");
$input->setProp('log', new TempFile);
$input->getProp('log')
      ->setProp('filename', "FILE_PATH");
echo base64_encode(serialize($input));
```

Listing 2: Example of a payload template

The fuzzer generates a property tree and converts this tree into an injection object by creating a PHP file that instantiates the object. In this PHP file, FUGIO defines all the classes used in this tree, sets property values, and serializes the created class, as shown in Listing 2. By executing this file, the fuzzer generates the input, which is fed into the PUT.

Note that the fuzzer does not inject the defined classes into this PHP file. These classes are helper classes and are only designed for generating a serialized injection string. The injected serialized string actually exploits existing classes in a target PHP application.

Mutation. The fuzzer mutates the property tree of the selected seed (Line 6). The fuzzer visits each property in the property tree and checks its type. When this type is `Object`, the fuzzer randomly selects one class from the candidate classes of this property. Otherwise, the fuzzer randomly selects one PHP type among *string*, *integer*, *boolean*, *file*, *array*, and *reference*.

For *string*, *integer*, *boolean*, and *file* types, the fuzzer assigns a random value of the selected type to this property. For an *array* property, the fuzzer randomly sets up the array size and assigns random values to keys and values in the array. For a *reference* property, the fuzzer identifies its owner and its other properties that the static and dynamic analyzer compute. It then randomly selects one of the other properties and assigns its reference to the target *reference* property. After mutating properties, the property tree is converted into the PHP file, and the fuzzer executes the instrumented PUT with the generated input from this file (Line 7).

Feedback. The fuzzer conducts a feedback-driven targeted fuzzing that leverages the execution result of a given input to derive new seeds, which are more promising candidates to reach the sensitive sink in the given POP chain. There exist four types of feedback that the fuzzer leverages: 1) branch coverage, 2) the depth of a gadget reached, 3) property hinting, and 4) reference error.

For branch coverage, the fuzzer adds a mutated seed to the seed pool when this mutated input covers new branches (Lines 9–10). For the second type of feedback, the fuzzer leverages a gadget depth. When the execution of a mutated seed enables the execution of a new gadget in the POP chain that is deeper than that the original seed of this mutated one reached, the fuzzer adds this mutated seed to the seed pool with the updated depth of the reached gadgets (Lines 17–18).

When observing a property that is used in a conditional statement with a comparison operator or specific built-in type-checking functions, such as `is_string()`, `is_int()`, and `is_array()`, the fuzzer stores the inferred type or a constant operand in this conditional statement for the property. For each hinted property, it generates a hinted seed in which the value of the hinted property is set as the inferred value or is randomly mutated by the inferred type (Lines 19–21).

Furthermore, the fuzzer observes reference errors in the invocation of a method call leveraging a receiver (e.g., `$receiver->method()`). When an observed error is due to a missing property or an incorrect object in the receiver, the fuzzer appends the missing property node in the tree of the current input or assigns an `Object` with a value chosen from among the classes that have the target method call name (Lines 22–24).

Exploit oracle. The fuzzer leverages the exploit oracle to determine whether a generated input object is able to exploit the POI vulnerability. When the mutated seed has reached the sensitive sink, the fuzzer reports that the given POP chain is *probably exploitable* with the generated input (Lines 11–12). To confirm the exploitability of an object identified as probably exploitable, the fuzzer checks whether a generated input can control the actual arguments of a sensitive sink. The oracle compares each property value in the input object with the sink arguments, thereby deriving a set of candidate properties into which the fuzzer injects payloads. For each candidate, the fuzzer sets its property value to an attack payload that depends on the target sensitive sink. For instance, if the type of sensitive sinks is *echo*, which can allow an XSS attack, the attack payload is set to `<script>alert(1);</script>`. In the case that sensitive sinks cause file deletion, we set the attack payload to an existing file path.

Lastly, the fuzzer checks whether the injection object invokes the sensitive sink with the attack payload in its argument. FUGIO hooks 26 sensitive sink calls in PHP and checks whether each sink is invoked with an actual parameter containing the attack payload. If so, the fuzzer reports that the given POP chain is *exploitable* with the generated payload and terminates the fuzzing campaign (Lines 13–15). Note that Listing 2 shows a final output, a PHP snippet that defines a generated exploit object and prints the serialized string of this object. FUGIO is also able to generate an attack HTTP(S) request by filling this serialized string into a GET, POST, or COOKIE input parameter in the original request that the detector found to trigger the POI vulnerability (§5.1).

In summary, the fuzzer generates an attack string and executes the PUT with this input, which is later deserialized into an exploit object. The exploit oracle checks whether this injection object invokes the series of gadgets in the given POP chain and invokes the sink function with the attack payload.

Manager. FUGIO runs the POP chain identifier and the POP chain fuzzer in parallel. Once the POP chain identifier computes a set of chains while visiting each sensitive sink, FUGIO manages fuzzing campaigns by invoking the fuzzing process for each chain. Therefore, while the POP identifier computes POP chains for a sink, FUGIO can initiate fuzzing on the chains generated from a different sink.

FUGIO prioritizes POP chains with a shorter length when selecting POP chains to fuzz. We implemented this fuzzing policy to favor short POP chains when generating exploit objects. In this way, FUGIO terminates the fuzzing process for the corresponding sink when an exploit object in short chains is found before it attempts to check long chains, which requires more computation resources. We set the ratio of CPU cores to execute the POP chain identifier and the fuzzer to be 3 to 1. After the POP chain identifier is done, all CPU cores are assigned to the fuzzer.

6 FUGIO Implementation

FUGIO is implemented in 20K+ LoC of Python and PHP. When identifying object injection points by hooking PHP built-in functions, we used *runkit* [14] and *uopz* [34]. However, since these extensions do not support hooking `eval`, we implemented this functionality to extract dynamically generated functions and classes. To incorporate this hooking functionality, we prepended the PHP file implementing hooking using a `.htaccess` file before executing every PHP file.

We also leveraged *PHP-Parser* [45] when parsing PHP files into ASTs for the static analysis and instrumenting a PUT. For communicating between different modules, we used *RabbitMQ* [63]. To support open science and further research, we release FUGIO at https://github.com/WSP-LAB/FUGIO.

7 Evaluation

We evaluate FUGIO by measuring its efficacy in generating exploit objects (§7.2) and comparing its performance with a previous study and an open-source tool (§7.3). We then demonstrate the degree to which several parameters affect the generation of functional exploits (§7.4). We also evaluate FUGIO in finding previously unreported POI vulnerabilities as well as their exploits (§7.5). We demonstrate case studies with reported exploit objects (§7.6).

7.1 Experimental Setup

We evaluated FUGIO on 30 PHP applications. For each app, we prepared a known POI vulnerability, thus tasking FUGIO with generating exploit objects that trigger the vulnerability causing file deletion/modification/creation, command injection, or remote code execution. Of 30 applications, eight applications were the same as those that Dahse *et al.* used for evaluation [13]. We also included 21 applications from PHPGGC [2]. In PHPGGC, 12 packages are PHP libraries. Thus, we made a simple PHP application using each library and injected a POI vulnerability into this application. For the remaining nine applications, we leveraged known POI vulnerabilities: CVE-2018-20148, CVE-2019-6339, and [64]. The selection criteria for these benchmarks are as follows: 1) the vulnerable versions of applications are still accessible, and 2) the sizes of vulnerable applications are not trivial.

Environment. We performed the evaluation on a Linux workstation equipped with two Intel Xeon Gold 6238 CPUs @ 2.10 GHz and 384 GB of RAM. For FUGIO execution, we prepared a Docker container for each PHP version and installed web applications according to their corresponding versions.

7.2 Performance of FUGIO

We conducted five fuzzing campaigns for each of the 30 applications. Each fuzzing campaign lasted 12 hours. For each chain, we set FUGIO to conduct fuzzing for at most 100 seconds and to assemble at most seven gadgets. We evaluate the FUGIO's sensitivity on these parameters in §7.4.

Table 1 summarizes the evaluation results. The third column represents the number of identified POP chains, and the fourth column represents the number of POP chains on which fuzzing was conducted to generate exploits. The *Covered sinks* column represents the number of unique sensitive sinks in the detected POP chains.

The *Exploitable chains* column represents the number of exploitable POP chains, which means that FUGIO is able to generate PHP object exploits. The *Probably exploitable chains* column represents the number of chains with generated input objects that succeeded in reaching the sink function in each chain, but FUGIO could not confirm its exploitability because the generated exploit failed to pass the exploit oracle. Note that each cell in these columns represents a median value from five fuzzing trials; the minimum and maximum values are in square brackets. Also, the number in parentheses represents the number of reported unique chains during the five fuzzing trials. The *True positive chains* column represents the number of exploitable chains that we manually confirmed. The numbers to the left and right of the plus sign represent true positive chains from exploitable (E) and probably exploitable (PE) chains, respectively. The sum of these two numbers yields the number of total exploitable objects that FUGIO created during the five fuzzing trials. The last two columns show the time spent in identifying POP chains and running FUGIO, respectively.

From the 30 applications, FUGIO reported a total of 68 E chains. We manually verified these E chains and confirmed that FUGIO yielded no false positives. Among the 66 PE chains, 26 chains were indeed exploitable. *FUGIO generated exploit objects from each of the 27 vulnerable applications.*

FUGIO did not succeed in generating exploits for GLPI, Vanilla, and Yii. In GLPI, FUGIO did not find any E chains because it actually has no exploitable chains. This means that the attacker is able to inject an input object but unable to exploit the vulnerability in GLPI. In Vanilla, FUGIO missed one exploitable chain that triggers an LFI vulnerability. This is because FUGIO does not support LFI sinks for computing chains. In Yii, FUGIO identified exploitable chains, but the fuzzer could not generate an exploit object for them. The generation of this exploit requires assigning an array value holding an appropriate index and its corresponding class object to a property of the exploit object. FUGIO was unable to generate an exploit object satisfying these conditions within a fuzzing timeout.

Note that FUGIO identified approximately 10 million POP chains that triggered 1,637 sensitive sinks. Identified chains ranged from 0 (Vanilla) to 3.2 million chains (Joomla). These statistics demonstrate the need to automate the generation of exploit objects.

We further analyzed why FUGIO reported 26 exploitable chains as PE chains. 1) FUGIO does not perform the exploit oracle on PE chains with sinks that take file resources, such as `fwrite`, because the attacker cannot inject a file resource as a serialized string. However, we found that nine PE chains were exploitable by leveraging an existing `fopen` callsite with an injected file name. 2) The exploit oracle only reports E chains when an injected payload appears in an actual argument of the target sink without any loss. However, four chains were exploitable even in the case that the attacker can partially inject an attack string into the parameter. For the remaining 13 chains, FUGIO was unable to pinpoint object properties to inject attack payloads within a given fuzzing timeout.

Table 1: Evaluation of FUGIO: The number of exploitable and probably exploitable chains represents the number of functional exploit objects that FUGIO generates (WP: WordPress).

PHP Version	Application	Identified Chains	Fuzzed Chains	Covered Sinks	Exploitable Chains	Probably Exploitable Chains	True Positive Chains	Chain Analysis Time	Total Time
PHP 5.4	Contao CMS	26,180	25,732	41	3 [3 – 3] (3)	2 [2 – 7] (8)	3 + 2	117m 49s	Timeout
	Piwik	445,384	14,739	40	0 [0 – 0] (0)	0 [0 – 4] (4)	0 + 1	407m 24s	Timeout
	GLPI	544,776	8,898	8	0 [0 – 0] (0)	0 [0 – 0] (0)	0 + 0	Timeout	Timeout
	Joomla	3,292,647	9,075	47	0 [0 – 1] (1)	0 [0 – 5] (5)	1 + 0	Timeout	Timeout
	CubeCart	30	28	11	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	7s	1m 51s
	CMS Made Simple	16	16	13	1 [0 – 1] (1)	0 [0 – 1] (0)	1 + 0	6s	1m 49s
	Open Web Analytics	886	765	18	11 [9 – 12] (12)	5 [4 – 6] (5)	12 + 4	45s	16m 52s
	Vanilla Forums	0	0	0	0 [0 – 0] (0)	0 [0 – 0] (0)	0 + 0	12s	12s
	SwiftMailer 5.0.1	13,387	12,891	16	1 [0 – 2] (2)	1 [0 – 2] (3)	2 + 2	5m 28s	289m 30s
	SwiftMailer 5.1.0	14,875	14,875	16	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	7m 45s	330m 37s
	Smarty	15	15	7	0 [0 – 1] (1)	3 [2 – 3] (2)	1 + 0	4s	1m 57s
	ZendFramework	1,271,410	8,732	264	0 [0 – 1] (1)	1 [0 – 5] (8)	1 + 0	Timeout	Timeout
PHP 5.6	PHPExcel 1.8.1 (w/ WP)	2,787	2,679	37	3 [3 – 3] (4)	1 [1 – 1] (2)	4 + 0	1m 31s	55m 49s
	PHPExcel 1.8.2 (w/ WP)	3,333	2,677	37	3 [3 – 3] (3)	1 [1 – 1] (1)	3 + 0	1m 56s	56m 10s
	Dompdf (w/ WP)	639,904	8,350	115	0 [0 – 1] (1)	0 [0 – 0] (0)	1 + 0	Timeout	Timeout
	Guzzle (w/ WP)	80,285	27,948	43	3 [0 – 3] (3)	1 [0 – 1] (1)	3 + 1	47m 20s	Timeout
	WooCommerce 2.6.0 (w/ WP)	3,857	3,747	28	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	23s	76m 18s
	WooCommerce 3.4.0 (w/ WP)	158,636	12,004	27	0 [0 – 1] (1)	0 [0 – 0] (0)	1 + 0	311m 56s	581m 25s
	Emailsubscribers (w/ WP)	1,844	1,793	28	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	13s	37m
	EverestForms (w/ WP)	2,081	2,032	25	1 [1 – 1] (1)	0 [0 – 0] (0)	1 + 0	14s	42m 4s
PHP 7.2	TCPDF	2	2	2	2 [2 – 2] (2)	0 [0 – 0] (0)	2 + 0	4s	13s
	Drupal7	15	15	11	1 [0 – 1] (1)	4 [4 – 5] (4)	1 + 0	7s	2m 2s
	SwiftMailer 5.4.12	14,977	14,977	16	1 [1 – 1] (1)	2 [1 – 3] (5)	1 + 4	5m 7s	330m 16s
	SwiftMailer 6.0.0	13,495	13,175	16	1 [1 – 2] (3)	1 [0 – 3] (5)	3 + 4	4m 45s	286m 25s
	Monolog 1.7.0	147	144	34	1 [0 – 2] (6)	0 [0 – 1] (1)	6 + 1	3s	3m 36s
	Monolog 1.18.0	4,525	4,230	55	2 [2 – 4] (8)	1 [0 – 2] (4)	8 + 4	12s	84m 16s
	Monolog 2.0.0	11,842	11,701	17	0 [0 – 0] (0)	0 [0 – 1] (1)	0 + 1	2m 36s	262m 2s
	Laminas	3,254	3,254	4	2 [2 – 2] (2)	0 [0 – 0] (0)	2 + 0	3m 18s	68m 1s
	Yii	2,428,535	9,033	453	0 [0 – 0] (0)	0 [0 – 2] (2)	0 + 0	Timeout	Timeout
	TYPO3	1,073,189	8,751	208	0 [0 – 5] (7)	0 [0 – 6] (7)	7 + 2	Timeout	Timeout
Total		10,052,313	222,278	1,637	68	66	68 + 26		

7.3 Comparison to State-of-the-Art Tools

We compared the performance of FUGIO to the performance that Dahse *et al.* reported in §7.3.1 and that of PHPGGC in §7.3.2. Dahse *et al.* proposed a static tool that reports exploitable chains [13]. On the other hand, FUGIO reports not only exploitable chains but also their exploit objects. In §7.3.1, we conduct a fine-grained comparison on the FUGIO's ability to identify exploitable chains that Dahse *et al.* reported.

PHPGGC is an open-source tool that generates PHP exploit objects only for known exploit chains in specific versions of PHP applications [2]. Because PHPGGC lists known exploitable POP chains, we used these chains to measure false negatives that FUGIO produces in §7.3.2. Note that, unlike PHPGGC, FUGIO is a general AEG tool for assembling gadgets and reporting exploit objects for any PHP applications.

Experimental setup. For each application, we ran FUGIO five times with a timeout of 12 hours. We set FUGIO to conduct fuzzing for at most 100 seconds for each chain. In §7.3.1, we set FUGIO to identify POP chains with a length of less than nine, which Dahse *et al.* used. In §7.3.2, we considered chains of which length is at most nine, which is the maximum exploitable chain length listed in PHPGGC.

7.3.1 Comparison to Dahse *et al.*

Target classes. Dahse *et al.* analyzed whether an object's class for injection is loadable by statically checking a stack of included files [24]. However, when there exists at least one autoloader callback in an application, they assume that all existing classes are loadable, which is no longer a valid assumption because this bug was patched in PHP 5.4.24 and

Table 2: The number of exploitable POP chains and their exploit objects that FUGIO and Dahse *et al.* reported.

Tool	Contao	Piwik	GLPI	Joomla	CubeCart	CMSMS	OWA	Vanilla	Total
FUGIO	11	1	0	2	1	1	17	0	33
[13]†	14	3	0	3	1	1	5	0	27

† Excludes POP chains conducting SQLi, XXE, and LFI attacks.

5.5.8 [12, 59, 60]. On the other hand, FUGIO checks loadable classes by dynamically invoking all existing autoloader callbacks to check their availability. To conduct a fair comparison, we consider all user-defined classes as available gadgets when at least one autoloader exists, making the same assumption for available gadgets.

Sensitive sinks. Dahse *et al.* classified the detected chains into six vulnerability types: file deletion (FD), file creation (FC), file modification (FM), SQL injection (SQLi), local file inclusion (LFI), and XML external entity injection (XXE). We specified a total of 26 sensitive sinks to generate exploits for FD, FC, FM, and command injection vulnerabilities, as listed in Appendix 11.1. We excluded sinks enabling LFI and XXE because those vulnerabilities are not reproducible in PHP 5.4 [61, 62]. For SQLi, it is possible to identify POP chains; however, the fuzzer cannot reach the sensitive sink since such chains require a database account or an instance that is already connected to the database. We leave this to future work.

Table 2 shows the number of *E* chains that FUGIO identified and successfully generated their exploit objects, compared to the number of exploitable chains that Dahse *et al.* reported. FUGIO generated exploit objects for 33 *E* chains from six applications while Dahse *et al.* identified 27 *E* chains from the same applications. Since Dahse *et al.* omitted the details of each chain, we could not match each exploitable chain. Thus, we compare the numbers of exploit objects that FUGIO reported with the numbers reported in their paper.

For the six missing *E* chains from Contao, Piwik, and Joomla, FUGIO could not generate exploit objects due to the following reasons: 1) four chains required passing complicated conditions to reach the sink functions or to control the arguments of these sink functions; and 2) two chains required specific OS environments and the existence of certain files or directories to reach the sink functions. These limitations stem from the unsound nature of fuzz testing; we believe that more advanced fuzzing optimization and computation resources will decrease the number of false negatives.

Note that the static approach of Dahse *et al.* reported 10 false positives due to statically unresolved callsites impeding their taint data flow analysis [13]. This means that auditors should check the exploitability of all reports, including these false positives. By contrast, FUGIO reported no false positives since it executes a PUT with a generated object and confirms the exploitability using the exploit oracle.

Table 3: Comparison of exploitable POP chains found by FUGIO and listed in PHPGGC (WP: WordPress)

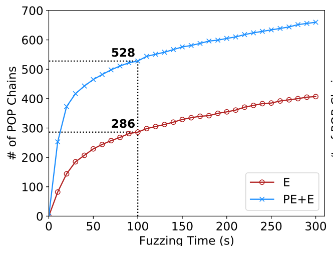
Applications	PHPGGC	FUGIO		
	Known Chains	Detected Chains	Exploits	New Exploits
SwiftMailer 5.0.1	1	1	0	1
SwiftMailer 5.1.0	1	1	0	3
Smarty	2	1	1	0
ZendFramework	4	2	1	0
PHPExcel 1.8.1 (w/ WP)	5	5	5	1
PHPExcel 1.8.2 (w/ WP)	5	5	5	1
Dompdf 0.8.0 (w/ WP)	1	1	0	0
Guzzle (w/ WP)	5	4	2	2
WooCommerce 2.6.0 (w/ WP)	1	1	1	0
WooCommerce 3.4.0 (w/ WP)	1	1	1	0
Emailsubscribers (w/ WP)	1	1	1	0
EverestForms (w/ WP)	1	1	1	0
TCPDF 6.3.2	1	1	1	1
Drupal7	2	1	1	1
SwiftMailer 5.4.12	1	1	1	6
SwiftMailer 6.0.0	1	1	0	11
Monolog 1.7.0	2	2	0	1
Monolog 1.18.0	1	1	0	3
Monolog 2.0.0	1	1	0	0
Laminas	1	1	1	1
Yii	1	1	0	0
Total	39	34	22	32

7.3.2 Comparison to PHPGGC

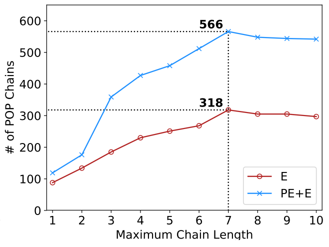
Additional setup. FUGIO terminates fuzzing for a sink when the fuzzer generates at least one exploit (§5) in order to explore diverse chains that invoke other sinks. However, PHPGGC lists multiple chains that share the same sink. Therefore, we set the fuzzing process not to terminate before fuzzing all enumerated chains for the corresponding sink.

Table 3 shows the experimental results on the capability of FUGIO generating exploit objects in 21 PHPGGC applications. The second column in the table represents the number of known exploitable chains that PHPGGC lists. The third column shows the number of gadget chains that FUGIO identified, and the fourth column represents the number of generated exploits for these identified chains. The last column shows the number of new exploitable chains that FUGIO found, along with their exploit objects. For the 39 exploitable chains in PHPGGC, FUGIO reported 34 chains and 22 exploit objects for these chains. Additionally, FUGIO found 32 new exploitable chains and their exploit objects, producing a total of 54 exploitable chains and their exploit objects.

Among a total of 39 chains in PHPGGC, FUGIO identified 34 POP chains; we analyzed the root causes for the five false negatives. In ZendFramework, Guzzle, and Drupal7, FUGIO did not find four chains because these applications used reflective calls to connect two gadgets. For example, when connecting the next gadget from the third gadget in a Zend-



(a) Fuzzing timeout



(b) Maximum chain length

Figure 5: The number of POP chains generated while varying the fuzzing timeout and maximum chain length (E: exploitable, PE: probably exploitable).

Framework chain, `$view->$helper` in the third gadget is supposed to call `Zend_Cache_Frontend_Function::call`. The target callee from this invocation is statically undecidable unless abstracting possible string values for `$helper`. Note that identifying a target callee from a reflective callsite remains an unresolved problem in the field of static analysis. Any other static approaches, including Dahse *et al.* [13], would have produced the same results. An alternative is to consider all gadgets from this callsite; however, this policy results in producing a prohibited number of chains.

The one remaining false negative leverages PHP built-in gadgets, which are not within our scope; FUGIO focuses on assembling user-defined gadgets in target applications.

Of the 34 identified chains, FUGIO generated exploit objects for 22 *E* chains. The missing chains were due to two reasons. 1) Nine chains were required to build an exploit object, several properties of which had array values holding other objects. Because such objects required FUGIO to assign to an object property an array value holding an appropriate index and its corresponding object, FUGIO could not generate these complex objects within a given fuzzing timeout. 2) For the three remaining chains, FUGIO did not conduct fuzz testing due to an insufficient time budget for testing other chains. When conducting fuzz testing on these three chains, FUGIO produced functional exploits.

We emphasize that FUGIO reported new 32 *E* chains that PHPGGC does not list. These results demonstrate that FUGIO is capable of helping developers find exploitable chains via AEG, which may have been missed due to difficult property-oriented programming.

7.4 Hyperparameters

We evaluate the effectiveness of two parameters in FUGIO: fuzzing timeout and max chain length.

Fuzzing timeout. The fuzzer executes a PUT with generated inputs until it reports an exploit object. The longer the fuzzer runs, the more exploits can be generated. However, we set the fuzzing timeout to test other POP chains within a given time budget. In this section, we conducted an experiment to determine how long the fuzzer should run for each POP chain.

To build an evaluation dataset, we sampled POP chains to fuzz. For each sensitive sink in the 30 applications, we sampled a maximum of 10 chains of which the last gadget has this sink; the total of sampled POP chains was 13,582.

For each sampled chain, we ran the fuzzer five times with a timeout of 300 seconds. The fuzzer generated a total of 408 *E* chains and 662 (*PE*+*E*) chains. For each *PE* or *E* chain, we measured the time that the fuzzer spent generating exploit objects. We then counted the number of chains for which the exploit object was generated within a certain specified time, from 0 to 300 seconds in increments of 10 seconds. We plotted the results in Figure 5a. We then computed the percentage of chains for which the fuzzer generated the exploit object within the specified time out of the total chains for which the exploit object was generated within 300 seconds. We then set the fuzzing timeout as the time when the percentage becomes greater than 70%. For the exploitable chains, the percentage is 70.1% at 100 seconds, and for the probably exploitable and exploitable chains, the percentage is 70.2% at 50 seconds. Therefore, we set the fuzzing timeout to be 100 seconds in all experiments (§7).

Chain length. When identifying POP chains, the chain identifier takes the height of a call tree as a parameter (§5.4). The height means the maximum length of POP chains that FUGIO computes. We conducted an experiment to measure the efficacy of the maximum chain length in reporting exploit objects.

The method for preparing POP chains for this evaluation is similar to the aforementioned experiment involving fuzzing timeout. We set the maximum chain length to a value from 1 to 10 and sampled POP chains for each condition. For example, if the maximum chain length was set to 3, sampling will be performed among chains of lengths 1, 2, and 3.

FUGIO conducted five fuzzing campaigns for the sampled chains, using a timeout of 100 seconds. We computed totals of *E* chains and both *PE* and *E* chains generated for each maximum chain length. Figure 5b shows the experimental results. When changing the maximum chain length from 7 to 8, the number of chains decreases. This is because a fuzzing timeout budget curtails the fuzzing of all promising chains to generate exploit objects. Therefore, we set the maximum chain length to be seven in §7.2.

7.5 Field Test

We evaluated the efficacy of FUGIO in the latest version of two PHP applications: WordPress 5.4.2 with WooCommerce and Concrete5 8.5.4. WooCommerce is one of the most popular WordPress plugins over five million active installations [3].

In WordPress, FUGIO identified 39 POP chains for seven sensitive sinks and reported one exploitable chain with an exploit object. The chain starts with `WC_Log_Handler_File::__destruct` and triggers `call_user_func` with two user-

```
 1  <?php
 2  class WC_Log_Handler_File extends WC_Log_Handler {
 3    function __destruct() { // 1st Gadget
 4      foreach ($this->handles as $handle) {
 5        if (is_resource($handle)) {
 6          fclose($handle);
 7  } } } }
 8  class Requests_Utility_FilteredIterator extends
 9        ArrayIterator {
10    public function current() { // 2nd Gadget
11      $value = parent::current();
12      $value = call_user_func($this->callback,
13                              $value); // Sink
14      return $value;
15  } }
16
17  // Generate injection object
18  $obj = new WC_Log_Handler_File;
19  $obj->setProp('handles',
20                new Requests_Utility_FilteredIterator(
21                  ["args" => "ARGUMENT_OF_CALLBACK"]
22                ));
23  $obj->getProp('handles')
24      ->setProp('callback', "CALLBACK_TO_BE_CALLED");
25
26  // Trigger POI vulnerability
27  $data = unserialize($input);
```

Listing 3: The exploitable POP chain of WordPress

controllable arguments. Therefore, the attacker can remotely invoke any existing functions with her choices of arguments. For instance, the attacker invokes `system` with a forged argument, conducting a shell command injection attack.

In Concrete5, FUGIO found four exploitable chains with exploit objects among 5,016 chains for 201 sensitive sinks. Three chains and their exploitable objects enable the attacker to delete an arbitrary file and the one remaining chain with its exploit object allows the attacker to invoke any user-defined functions with her choices of arguments.

Disclosure. We reported the vulnerabilities with exploit objects to HackerOne [21]. The WordPress team notified that the vulnerability was reported in September 2018, but has remained a non-disclosed bug. The Concrete5 team patched the reported vulnerability and assigned CVE-2021-40102.

7.6 Case Studies

We introduce two *E* chains and their exploit objects that FUGIO reported from WordPress 5.4.2 with WooCommerce and the latest version of Concrete5.

WordPress with WooCommerce. Listing 3 shows a POP chain and its exploit object that FUGIO generated when detecting the WordPress vulnerability in §7.5. Its exploitable POP chain consists of two gadgets, `WC_Log_Handler_File::__destruct` and `Requests_Utility_FilteredIterator::current` that invokes the `call_user_func` sink function.

The `current` method of a class that inherits `ArrayIterator` is invoked by any call to `foreach` on that class's object. When the `handles` property in Line 4 is set to the `Requests_Utility_FilteredIterator` instance, its `current` method is invoked.

The fuzzer starts by generating an exploit object for this

```
 1  <?php
 2  namespace simplehtmldom_1_5 {
 3  class simple_html_dom {
 4    function __destruct() { // 1st Gadget
 5      $this->clear();
 6    }
 7    function clear() { // 2nd Gadget
 8      if (isset($this->parent)) {
 9        $this->parent->clear();
10        unset($this->parent);
11  } } } }
12  namespace Stash\Driver {
13    class FileSystem extends AbstractDriver {
14      public function clear($key = null) { // 3rd Gadget
15        $path = $this->makePath($key);
16        if (is_file($path)) {
17          $return = true;
18          unlink($path); // Sink
19      } }
20      protected function makePath($key = null) {
21        if (!isset($this->cachePath)) {
22          throw new LogicException ('Error');
23        }
24        $basePath = $this->cachePath;
25        $path = $basePath;
26        return $path;
27  } } }
28
29  // Generate injection object
30  $obj = new simplehtmldom_1_5\simple_html_dom;
31  $obj->setProp('parent', new Stash\Driver\FileSystem);
32  $obj->getProp('parent')
33      ->setProp('cachePath', "FILE_TO_DELETE");
34
35  // Trigger POI vulnerability
36  $data = is_dir($input);
```

Listing 4: The exploitable POP chain of Concrete5

chain by picking the first gadget class, `WC_Log_Handler_File` (Line 18). To connect the next gadget, the fuzzer mutates the value of each property declared in `WC_Log_Handler_File` class. When the `handles` property in Line 4 is mutated as `Object` type and its value is set to `Requests_Utility_FilteredIterator` (Lines 19–22), the `current` method is invoked and the fuzzer reaches the sink function `call_user_func` in Line 13. Then, the fuzzer checks whether the mutated property can control the arguments of this sensitive sink. The first argument of `call_user_func` is the `callback` property of `Requests_Utility_FilteredIterator` class, which should be set to an attacker's choice of callback (Lines 23–24). This callback's argument can be passed through `$value`, the second argument of `call_user_func`, which represents the value of an item that is currently iterated (Line 21). Since the attacker is able to control the first argument of this `call_user_func` sink, FUGIO concludes fuzz testing by reporting this exploit object.

Concrete5. Listing 4 shows a POP chain and its exploit object that FUGIO reported from Concrete5 (§7.5). The chain length is three. This chain consists of `simple_hmtl_dom::__destruct`, `simple_hmtl_dom::clear`, and `Filesystem::clear` that invokes the `unlink` function.

To generate an exploit object for this chain, the fuzzer picks the first gadget class, `simple_hmtl_dom` (Line 30). The first gadget, `simple_hmtl_dom::__destruct`, unconditionally calls the second gadget, `simple_hmtl_dom::clear`.

To connect the third gadget, `Filesystem::clear`, the condition in Line 8 should be passed. Thus, the fuzzer will assign a random value to the `parent` property. After passing this condition, the fuzzer will encounter a reference error in Line 9 when attempting to invoke `clear` with the `parent` property. Now, the fuzzer assigns a class object that has the `clear` method. When the fuzzer selects `FileSystem` for this class object, the last gadget will be executed (Line 31). To reach the sensitive sink, the `$path` variable should be a file path (Line 16). Although the fuzzer does not know how to set `$path`, it can be properly set while mutating the properties of `FileSystem`. The fuzzer attempts to assign a random value in the `cachePath` property. By setting `cachePath` as a file to delete (Lines 32–33), the fuzzer succeeded in generating an exploit object for the identified POP chain.

8 Discussion and Limitations

Security threats of deserializing an adversarial object have existed in not only PHP but also other programming languages, including Python [31, 41], Java [25, 68], Ruby [33], Android [48], and .NET [18, 43, 55]. Depending on available gadgets, an adversary conducts various malicious behaviors.

Each language has its own recommendations for mitigating this threat. A basic takeaway is not to directly deserialize data from untrusted sources [46]. The common practice of following this recommendation is to sanitize user inputs [17, 46]. Unfortunately, sanitization logic should differ based on target deserialization methods, which often leads to implementing incorrect sanitization checks [20, 54].

Another line of recommendation suggests that developers permit only primitive data types like JSON (e.g., `json_decode` in PHP and `json.loads` in Python), YAML (e.g., `SnakeYAML` in .NET, and `PyYAML` in python), or XML (e.g., `XMLDecoder` in JAVA), which do not invoke deserialization callbacks [46, 53]. Unfortunately, Muñoz and Mirosh [43] found that many JSON libraries in .NET and Java were exploitable because they invoked setters to populate object fields. Other works [16, 55] have reported similar vulnerabilities in XML and YAML formats.

Whitelisting or blacklisting classes to deserialize is a passive mitigation method [39]. This approach restricts which classes are allowed or disallowed to be deserialized by leveraging the features supported by each language or by raising errors when deserializing blacklisted classes. However, this approach requires a significant engineering cost to specify allowed classes for (de)serialization [20].

Existing tools have focused on detecting unsafe deserialization [2, 8, 19, 42]. Burp Suite detects vulnerabilities by sending predefined payloads using [19] for Java and [2] for PHP applications [15, 51]. SerialDetector identifies unsafe deserialization using a taint dataflow analysis and validates identified vulnerabilities by generating payloads only for known gadget chains [55]. They do not assemble available gadgets to identify promising chains. Several static approaches have focused on identifying exploitable gadget chains [13, 22]. However, these studies require a manual examination to remove false positives. By contrast, FUGIO is a general AEG tool that identifies promising POP chains and generates exploits.

FUGIO has limitations. FUGIO only assembles gadgets extracted from target PHP applications, not gadgets from PHP internal classes. Therefore, it cannot generate exploits using PHP internal gadgets. Generating such exploits require the manual effort of explicitly providing internal gadgets in PHP to FUGIO. For the same reason, FUGIO is unable to leverage gadgets in PHP binary modules [49] to which PHP source code is converted

FUGIO is also unable to cover reflective calls of which target callee is statically undecidable when enumerating POP chains. Considering all existing gadgets for this target callee results in a prohibited number of chains to conduct fuzzing testing. A sophisticated static analysis that computes possible values for this target callee is one alternative approach for decreasing false negatives. Due to the nature of fuzz testing [38], finding exploits may require multiple campaigns or longer timeouts when a target chain has a large number of conditions.

9 Related Work

Finding vulnerabilities in web applications. There is a vast volume of studies on finding vulnerabilities in PHP applications. Huang *et al.* introduced *WebSSARI* to detect insecure information flow using a typestate-based static analysis algorithm [29, 30]. Xie *et al.* presented a three-tier analysis for capturing information at the intra-block, intra-procedural, and inter-procedural levels [66]. *Pixy* performed additional alias and literal analysis to provide more comprehensive and precise results [35, 36]. Son *et al.* presented static analysis techniques that identify semantic bugs [57] and remediate access-control bugs [56]. Backes *et al.* proposed an interprocedural analysis technique based on *code property graphs* that represent a program's syntax, control flow, and data dependencies in a single graph structure [6].

AEG. AEG has been used as a verification process that automatically checks whether the reported bug is security-critical. It contributes to eliminating false positives and helps developers prioritize bugs to patch [5]. In binary applications, AEG approaches primarily generate exploits by solving constraints that are combined 1) path constraints that a user input causes a given program to crash and 2) constraints for executing shellcode [4, 11, 23, 26, 27, 47, 65, 67].

AEG techniques are also applied in the analysis of web applications. Balzarotti *et al.* introduced *Saner* for validating the sanitization process by identifying a suspicious program path from input sources to sensitive sinks using static analysis and simulating the program with inputs containing attack strings using dynamic analysis [7]. Kieyzun *et al.*

and Huang *et al.* proposed AEG approaches based on concolic execution [28, 37]. Alhuzali *et al.* performed additional static analysis to construct a sequence of malicious HTTP requests that direct the execution of the program to a vulnerable sink [1].

Many symbolic execution studies have attempted to validate various types of vulnerabilities, such as cross-site scripting, SQL injection, or file inclusion, but no prior study has addressed generating exploits for POI vulnerabilities. Furthermore, those constraint solving approaches require considerable engineering efforts due to the necessity of modeling thousands of built-in functions. We chose to tackle this problem via fuzzing instead of symbolic execution.

10 Conclusion

We propose FUGIO, the first AEG tool for POI vulnerabilities. We present a series of static analyses, dynamic analyses, and fuzzing techniques to compute POP chains and generate exploits. FUGIO reported 68 exploit objects from 30 real-world PHP applications with known POI vulnerabilities. FUGIO also reported two previously unknown POI vulnerabilities with functional exploiting objects, demonstrating the efficacy of FUGIO in significantly alleviating laborious property-oriented programming burdens.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Xinyu Xing, for their helpful comments and suggestions to improve the paper. This work was supported by National Research Foundation of Korea (NRF) Grant No.: 2020R1C1C1009031 and partially supported by a Google Faculty Fellowship.

References

[1] Abeer Alhuzali, Rigel Gjomemo, Birhanu Eshete, and V.N. Venkatakrishnan. NAVEX: Precise and scalable exploit generation for dynamic web applications. In *Proceedings of the USENIX Security Symposium*, pages 377–392, 2018.

[2] Ambionics Security. PHPGGC: PHP generic gadget chains. `https://github.com/ambionics/phpggc`.

[3] Automattic. WooCommerce WordPress plugin. `https://wordpress.org/plugins/woocommerce/`.

[4] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[5] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[6] Michael Backes, Konrad Rieck, Malte Skoruppa, Ben Stock, and Fabian Yamaguchi. Efficient and flexible discovery of PHP application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 334–349, 2017.

[7] Davide Balzarotti, Marco Cova, Vika Felmetsger, Nenad Jovanovic, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Saner: Composing static and dynamic analysis to validate sanitization in web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 387–401, 2008.

[8] Moritz Bechler. Java Unmarshaller Security. `https://github.com/mbechler/marshalsec`.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, pages 30–40, 2011.

[10] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed greybox fuzzing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2329–2344, 2017.

[11] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 380–394, 2012.

[12] Johannes Dahse. Joomla! 3.0.2 POI (CVE-2013-1453) – Gadget Chains. `https://websec.wordpress.com/2014/10/03/joomla-3-0-2-poi-cve-2013-1453-gadget-chains/`, 2014.

[13] Johannes Dahse, Nikola Kreini, and Thorsten Holz. Code reuse attacks in PHP: Automated POP chain generation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 42–53, 2014.

[14] Dmitry Zenovich. Runkit (official PECL PHP runkit extension). `https://github.com/zenovich/runkit`.

[15] Federico Dotta. Java Deserialization Scanner (Burp Suite plugin). `https://github.com/federicodotta/Java-Deserialization-Scanner`.

[16] OWASP Stammtisch Dresden. JSON Deserialization Exploitation. `https://owasp.org/www-pdf-archive/Marshaller_Deserialization_Attacks.pdf.pdf`.

[17] Sondre Forland Fingann. Java deserialization vulnerabilities. Master's thesis, 2020.

[18] James Forshaw. Are you my type? breaking .NET through serialization. In *Proceedings of the Black Hat USA*, 2012.

[19] Christopher Frohoff. ysoserial. https://github.com/frohoff/ysoserial.

[20] Apostolos Giannakidis. The Deserialization Problem: What is the Deserialization vulnerability and what are the challenges in providing a solution. https://www.waratek.com/wp-content/uploads/2019/06/WP-Deserialization-20190610.pdf.

[21] HackerOne. Hacker-powered security testing & bug bounty. https://www.hackerone.com/.

[22] Ian Haken. Automated discovery of deserialization gadget chains. In *Proceedings of the Black Hat USA*, 2018.

[23] Sean Heelan. *Automatic generation of control flow hijacking exploits for software vulnerabilities*. PhD thesis, University of Oxford, 2009.

[24] Mark Hills, Paul Klint, and Jurgen Vinju. An empirical study of PHP feature usage: a static analysis perspective. In *Proceedings of the ACM International Symposium on Software Testing and Analysis*, pages 325–335, 2013.

[25] Philipp Holzinger, Stefan Triller, Alexandre Bartel, and Eric Bodden. An in-depth study of more than ten years of java exploitation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 779–790, 2016.

[26] Hong Hu, Zheng Leong Chua, Sendroiu Adrian, Prateek Saxena, and Zhenkai Liang. Automatic generation of data-oriented exploits. In *Proceedings of the USENIX Security Symposium*, pages 177–192, 2015.

[27] Shih-Kun Huang, Min-Hsiang Huang, Po-Yen Huang, Chung-Wei Lai, Han-Lin Lu, and Wai-Meng Leong. CRAX: Software crash analysis for automatic exploit generation by modeling attacks as symbolic continuations. In *IEEE International Conference on Software Security and Reliability*, 2012.

[28] Shih-Kun Huang, Han-Lin Lu, Wai-Meng Leong, and Huan Liu. CRAXweb: Automatic web application testing and attack generation. In *IEEE International Conference on Software Security and Reliability*, 2013.

[29] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Securing web application code by static analysis and runtime protection. In *Proceedings of the International Conference on World Wide Web*, pages 40–52, 2004.

[30] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, Der-Tsai Lee, and Sy-Yen Kuo. Verifying web applications using bounded model checking. In *International Conference on Dependable Systems and Networks*, pages 199–208, 2004.

[31] Insomnia Security. Deserialization, what could go wrong? https://insomniasec.com/cdn-assets/Deserialization_-__What_Could_Go_Wrong.pdf.

[32] Invicti. Acunetix. https://www.acunetix.com/.

[33] Luke Jahnke. Ruby 2.X Universal RCE Deserialization Gadget Chain. https://www.elttam.com/blog/ruby-deserialization/#content.

[34] Joe Watkins. uopz: User operations for zend. https://github.com/krakjoe/uopz.

[35] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 258–263, 2006.

[36] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Precise alias analysis for static detection of web application vulnerabilities. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, pages 27–36, 2006.

[37] Adam Kieyzun, Philip J Guo, Karthick Jayaraman, and Michael D Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of the International Conference on Software Engineering*, pages 199–209, 2009.

[38] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. Evaluating fuzz testing. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 2123–2138, 2018.

[39] Will Klieber. Prevent deserialization of untrusted data. https://wiki.sei.cmu.edu/confluence/display/java/SER12-J.+Prevent+deserialization+of+untrusted+data.

[40] Taekjin Lee, Seongil Wi, Suyoung Lee, and Sooel Son. FUSE: Finding file upload bugs via penetration testing. In *Proceedings of the Network and Distributed System Security Symposium*, 2020.

[41] Dan Lousqui. Explaining and exploiting deserialization vulnerability with python. https://dan.lousqui.fr/explaining-and-exploiting-deserialization-vulnerability-with-python-en.html.

[42] Alvaro Muñoz. ysoserial.net. https://github.com/pwntester/ysoserial.net.

[43] Alvaro Muñoz and Oleksandr Mirosh. Friday the 13th json attacks. In *Proceedings of the Black Hat USA*, 2017.

[44] Nergal. The advanced return-into-lib(c) exploits: PaX case study. http://phrack.org/issues/58/4.html.

[45] nikic. PHP-parser. https://github.com/nikic/PHP-Parser.

[46] OWASP. OWASP Top Ten 2017 A8: Insecure Deserialization. https://owasp.org/www-project-top-ten/2017/A8_2017-Insecure_Deserialization.

[47] Vartan A Padaryan, VV Kaushan, and AN Fedotov. Automated exploit generation for stack buffer overflow vulnerabilities. *Programming and Computer Software*, 41(6):373–380, 2015.

[48] Or Peles and Roee Hay. One class to rule them all: 0-day deserialization vulnerabilities in android. In *Proceedings of the USENIX Workshop on Offensive Technologies*, 2015.

[49] Phalcon. Zephir. https://github.com/zephir-lang/zephir.

[50] PortSwigger. Burp Suite. https://portswigger.net/burp.

[51] PortSwigger. PHP object injection slinger. https://github.com/portswigger/poi-slinger.

[52] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security*, 15(1):1–34, 2012.

[53] Ruby-Doc. Marshal. https://ruby-doc.org/core-2.6.3/Marshal.html.

[54] Sam Sanoop. SuiteCRM: PHAR deserialization vulnerability to code execution. https://snyk.io/blog/suitecrm-phar-deserialization-vulnerability-to-code-execution/.

[55] Mikhail Shcherbakov and Musard Balliu. SerialDetector: Principled and practical exploration of object injection vulnerabilities for the web. In *Proceedings of the Network and Distributed System Security Symposium*, 2021.

[56] Sooel Son, Kathryn S McKinley, and Vitaly Shmatikov. Fix Me Up: Repairing access-control bugs in web applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2013.

[57] Sooel Son and Vitaly Shmatikov. SAFERPHP: Finding semantic vulnerabilities in php applications. In *Proceedings of the ACM SIGSAC Workshop on Programming Languages and Analysis for Security*, 2011.

[58] SonarSource. PHP Code Quality and Code Security. https://www.sonarsource.com/php/.

[59] The PHP Group. PHP 5 changelog version 5.4.24. https://www.php.net/ChangeLog-5.php#5.4.24.

[60] The PHP Group. PHP 5 changelog version 5.5.8. https://www.php.net/ChangeLog-5.php#5.5.8.

[61] The PHP Group. PHP commit: add optional parameter to pass libxml document load options. https://github.com/php/php-src/commit/cb72e23c147c5a93161c24428762b434dc58524d.

[62] The PHP Group. Bug 62789: Autoloaders are invoked with invalid class names. https://bugs.php.net/bug.php?id=62789, 2012.

[63] VMware. Rabbitmq. https://www.rabbitmq.com/.

[64] WPScan. WooCommerce Authenticated Phar Deserialization. https://wpscan.com/vulnerability/9567f575-529d-4d66-980c-73cba6726673.

[65] Wei Wu, Yueqi Chen, Jun Xu, Xinyu Xing, Xiaorui Gong, and Wei Zou. FUZE: Towards facilitating exploit generation for kernel use-after-free vulnerabilities. In *Proceedings of the USENIX Security Symposium*, pages 781–797, 2018.

[66] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium*, pages 179–192, 2006.

[67] Luhang Xu, Weixi Jia, Wei Dong, and Yongjun Li. Automatic exploit generation for buffer overflow vulnerabilities. In *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security Companion*, pages 463–468, 2018.

[68] Yang Zhang, Yongtao Wang, Keyi Li, and Kunzhe Chai. New exploit technique in Java deserialization attack. In *Proceedings of the Black Hat EU*, 2019.

11 Appendix

11.1 Target Sensitive Sink Functions

We list the sink functions that FUGIO considers for each web attack as follows:

- File deletion: `unlink`, `rmdir`
- File creation: `fopen`, `fwrite`, `fputs`, `mkdir`, `copy`, `link`, `symlink`, `file_put_contents`
- File modification: `chmod`, `chown`, `chgrp`, `touch`
- Shell command injection: `popen`, `system`, `passthru`, `exec`, `proc_open`, `shell_exec`, `escapeshellcmd`
- Remote code execution: `eval`, `mail`, `call_user_func`, `call_user_func_array`, `preg_replace`