Code Reuse Attacks in PHP: Automated POP Chain Generation

Johannes Dahse, Nikolai Krein, and Thorsten Holz
Horst Görtz Institute for IT-Security (HGI)
Ruhr-University Bochum, Germany
{firstname.lastname}@rub.de

ABSTRACT

Memory corruption vulnerabilities that lead to control-flow hijacking attacks are a common problem for binary executables and such attacks are known for more than two decades. Over the last few years, especially *code reuse attacks* attracted a lot of attention. In such attacks, an adversary does not need to inject her own code during the exploitation phase, but she reuses existing code fragments (so called *gadgets*) to build a code chain that performs malicious computations on her behalf. *Return-oriented programming* (ROP) is a well-known technique that bypasses many existing defenses. Surprisingly, code reuse attacks are also a viable attack vector against web applications.

In this paper, we study code reuse attacks in the context of PHP-based web applications. We analyze how *PHP object injection* (POI) vulnerabilities can be exploited via *property-oriented programming* (POP) and perform a systematic analysis of available gadgets in common PHP applications. Furthermore, we introduce an automated approach to statically detect POI vulnerabilities in object-oriented PHP code. Our approach is also capable of generating POP chains in an automated way. We implemented a prototype of the proposed approach and evaluated it with 10 well-known applications. Overall, we detected 30 new POI vulnerabilities and 28 new gadget chains.

Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program analysis

General Terms

Security

Keywords

Static Code Analysis; Web Security; PHP Object Injection; Property-Oriented Programming; Code Reuse Attacks

CCS'14, November 3–7, 2014, Scottsdale, Arizona, USA.

ACM 978-1-4503-2957-6/14/11 ...$15.00.
http://dx.doi.org/10.1145/2660267.2660363.

1. INTRODUCTION

Memory corruption vulnerabilities, such as buffer overflows, format string bugs, and dangling pointers, are known for a long time and still constitute an intractable class of programming mistakes [37, 41]. While defense techniques such as *address space layout randomization* (ASLR) and *data execution prevention* (DEP) are widely deployed to hamper the exploitation of such vulnerabilities, an adversary can still utilize different techniques to circumvent such defenses. Especially *code reuse* techniques, such as for example *return-to-libc* [32], *return-oriented programming* (ROP) [27], and *jump-oriented programming* (JOP) [3], have received a lot of attention since they are able to bypass several kinds of security protections. With ROP and JOP, an attacker reuses available code fragments in memory (so called *gadgets*) and joins them together to construct the attack payload piece by piece (so called *gadget chains*) in scenarios where she cannot inject her own code.

In 2009, Esser showed that code reuse attacks are also viable in PHP-based web applications [9, 10]. More specifically, he introduced a similar exploitation approach for *object injection* vulnerabilities in web applications which abuses the ability of an attacker to arbitrarily modify the properties of an object that is injected into a given web application. Thus, the data and control flow of the application can be manipulated and he coined the term *Property-Oriented Programming* (POP). In the past five years, many *object injection* vulnerabilities were detected in popular open-source PHP software such as *Wordpress*, *Joomla*, and *Piwik*. They can lead to critical security vulnerabilities, such as *remote code execution*, and affect a majority of web servers since PHP is the most popular scripting language on the Web with a market share of more than 80% [43].

Similar to well-understood injection vulnerabilities such as *cross-site scripting* (XSS) [19] and *SQL injection* (SQLi) [13], *PHP object injection* (POI) vulnerabilities in a given application can be detected with the help of taint analysis. Broadly speaking, a vulnerability report is generated when untrusted user input reaches a security-sensitive sink [28]. Several analysis frameworks to detect different kinds of injection vulnerabilities were proposed in the last years [1, 18, 44, 46]. Recently, we presented a static code analysis approach that detects 20 types of injection vulnerabilities, including POI vulnerabilities [6]. However, no existing analysis tool is capable of deciding whether a given POI vulnerability is actually exploitable or not. This is a challenging analysis task since we need to identify a combination of gadgets in the code that allows an attacker to trigger another vul-

nerability by manipulating the control and data flow. Furthermore, complex object-oriented programming (OOP) features of PHP require a comprehensive analysis and—to the best of our knowledge—no existing static analysis tool for PHP-based web applications supports OOP analysis.

In this paper, we tackle these challenges and present the first automated approach to detect POP gadget chains to confirm POI vulnerabilities. By performing static code analysis that supports the analysis of PHP's OOP features, we are able to collect sensitive sinks in the application's code that can be reached after a PHP object was injected. More specifically, we propose an inter-procedural, field-sensitive, and object-sensitive data flow analysis that we can leverage to analyze the relevant OOP features. By analyzing the resulting path, we can also construct an actual attack payload for each detected gadget chain. The resulting chains allow us to verify the ability to exploit a potential POI vulnerability. We have implemented a prototype of the proposed analysis approach and tested it with 10 real-world applications vulnerable to PHP object injection. Besides confirming most of the previously reported POI in these applications in an automated way, our prototype reported several previously unknown POI vulnerabilities and gadget chains with only few false positives.

In summary, we make the following three contributions:

- We perform a systematic analysis of *PHP object injection* (POI) vulnerabilities and demonstrate how such vulnerabilities can be exploited via *Property-Oriented Programming* (POP), a variant of code reuse attacks against web applications.
- We are the first to propose an automated approach to statically detect POI vulnerabilities in object-oriented PHP code and to automatically verify the severity by constructing exploitable gadget chains.
- We evaluated our approach for 10 well-known applications recently affected by a *PHP object injection* vulnerability. As a result, we detected 30 new POI vulnerabilities and 28 new gadget chains.

2. PHP OBJECT INJECTION

A *PHP Object Injection* (POI) vulnerability occurs when unsanitized user input is used during the deserialization of data in a given application. PHP features so called *serialization* and *deserialization* functions that allow a programmer to store data of any type in an unified string format. This format makes it easy to transfer combined data structures and is often misused to create multidimensional cookies and similar data structures.

Since PHP allows deserialization of arbitrary objects, an attacker might be able to inject a specially prepared object with an arbitrary set of properties into the application's scope. Depending on the context, the attacker can trigger so called *magic methods* [39] and this potentially leads to a variety of vulnerabilities. Note that the type of vulnerability is highly dependent on the classes' implementation of their *magic methods*. Each *magic method* might call another (potentially security-relevant) PHP function (e.g., `eval()` or `fwrite()`) with attacker-controlled member variables as arguments that can lead to remote code execution, file inclusion, SQL injection, and any other kind of vulnerabilities.

We now introduce the concepts of *magic methods* (Section 2.1) and serialization (Section 2.2) in PHP. Both PHP features form the basis to exploit a POI vulnerability by utilizing *Property Oriented Programming*. This exploit technique combines both features and is described in Section 2.3. It is one of the most sophisticated attack techniques against PHP applications since it requires reusing already existing code in the application's classes.

2.1 Magic Methods in PHP

The concept of *object-oriented programming* (OOP) was considerably enhanced in version 5 of PHP and since then includes destructors, exceptions, interfaces, and further object-oriented concepts. OOP allows to logically encapsulate data and functionality in *objects*, while their implementation reside in the *class* definition. Each class can be initialized into an object that contains properties and methods that are defined in their designated class. These properties are called *attributes* (or *fields*), while *methods* describe a function accessible to an object.

Magic methods play an important role when exploiting POI vulnerabilities since they are *automatically* executed upon specific events. As we will see later on, they can be used to *start* a POP gadget chain. The following magic methods fulfill a special purpose and can be defined once per class:

- `__construct()`: This magic method implements the constructor inside a class that is called whenever a new object of that class is created. It is often used to initialize the object's attributes or to run other code before the object can actually be used.
- `__destruct()`: In contrast to the `__construct()` function, `__destruct()` is executed whenever the script terminates or the reference count of an object reaches zero. It is often used to invoke code that cleans up used data or terminates connections that were possibly established after the object was created.
- `__call()`: This function is always invoked when an inaccessible method of an object is called (e. g., `$obj ->invalid_method()`). It is handy in terms of error handling, since accessing invalid methods usually results in a fatal error and termination of the PHP application.
- `__callStatic()`: Similar to `__call()`, this magic function catches inaccessible calls in static context (e. g., `obj::invalid_method()`).
- `__get($name)`: The method `__get()` is automatically called when trying to *read* private, protected, or non-existent properties of an object. Since private and protected properties cannot be directly accessed from outside the object, the parameter `$name` is used to reference the desired property.
- `__set($name, $value)`: The method `__set()` is automatically called when trying to *write* to private or protected properties of an object. Because this is prohibited, this function allows the application to handle assignments such as `$obj->private = 'value'`.
- `__isset()`: Similar to previously mentioned methods, this function is called whenever `isset()` or `empty()` is used on a non-existent property.
- `__unset()`: Every time `unset()` is used on non-existent properties, this function is called with an argument

that describes the name of the variable that the application wants to unset.

- `__sleep()`: This magic method is triggered whenever an object is serialized. It gives the programmer the ability to let the object run any sort of cleanup-code before serialization.
- `__wakeup()`: In contrast to `__sleep()`, `__wakeup()` is called directly after deserialization. It is often used to reinitialize the application's state that was lost during serialization, for example the connection to a database.
- `__toString()`: Whenever an object is used in a string context (e.g., when it is concatenated with a string), this method is invoked to return a string representation of the object.
- `__invoke()`: This method is called whenever an object is used as a dynamic function name (e. g., `$obj()`).
- `__set_state($properties)`: Within an application, the function `var_export()` is used in order to display any sort of data as parsable PHP code. If an object is used as argument, the method `__set_state()` is called to define which properties are exported.
- `__clone()` This function is called when an object is cloned by the `clone` operator. It is equivalent to *copy-constructors* known in other languages. By implementing this method in a class, the programmer can specify what exactly should happen during cloning.

2.2 Serialization in PHP

PHP supports serialization and deserialization of all defined data types—including objects. Serialization is realized through the built-in function `serialize()` which accepts a single parameter and returns a serialized string that can be fed into `unserialize()` in order to retrieve said data again. This string is represented in an unified format which consists of several identifiers that specify the serialized data type. These identifiers have the following purpose:

- `a:` – defines that the passed parameter is an array. `a:` is always followed by a numerical value that specifies the size of the array.
- `i:` – simply defines a numerical value, e. g., `i:8;`.
- `b:` – specifies a boolean value, e. g., `b:0;` or `b:1;`.
- `s:` – defines a constant string. `s:` is always followed by a numerical value which declares the length of the string, e. g., `s:4:"test";`.
- `S:` – defines a constant string in encoded format.
- `O:` – represents an object in its serialized form. `O:` is followed by the length of the class name and by the name itself, e. g., `O:1:"A"`. It is then followed by the number of properties and the defined properties themselves. Note that a property can also consist of another object with its defined properties.

Further identifiers, such as `r:` and `R:`, exist that can be used to store references, but they are out of scope for our purpose. An example of the functionality behind PHP's serialization is given in the Listing 1. Line 2 serializes the array defined in line 1 and it therefore returns the string in line 4 which is then fed into `unserialize()` again. Line 6 then shows that the deserialization of the array returns the same values as they were previously defined.

```
1 $arr = array(1 => 2, 3 => "string");
2 $serialized = serialize($arr);
3 print $serialized . "\n";
4 // a:2:{i:1;i:2;i:3;s:6:"string";}
5 var_export(unserialize($serialized));
6 // array ( 1 => 2, 3 => 'string' )
```

Listing 1: Exemplary serialization of an array.

2.3 Property Oriented Programming

There are two preconditions that a PHP application needs to meet so that POP can be used to exploit a POI vulnerability. First, at least one *magic method* which gets called during the application's runtime needs to be defined in an object's class that the attacker wants to inject. Second, the chosen class needs to be loaded within the scope of the vulnerable `unserialize()` call the attacker passes her input to.

Each *magic method* can either be *context-dependent* or *context-independent*. Context-dependent means that an object has to be used in a certain way so that a *magic method* gets executed (see Section 2.1). Other *magic methods* are called *automatically* during the application's lifetime: the method `__wakeup()` and `__destruct()` is context-independent since `__wakeup()` is always called directly after deserialization of an object and `__destruct()` is always called once the object is destroyed. Both methods might contain suspicious code while using properties that can be arbitrarily defined when the object is deserialized.

Passing user input into the `unserialize()` function enables an attacker to inject specially crafted objects with chosen properties that will be used inside the *magic method*. However, when only context-dependent methods such as `__toString()` or `__call()` exist, the attacker has to choose a code path where the deserialized object is used accordingly to trigger the *magic method*. These code paths are often a lot more scarce and thus context-independent methods are a better choice for attacks.

Each *magic method* might also call different methods of other objects which are linked as members to the first object. In this scenario, it makes sense to check all other object methods, which can also be denoted as *gadgets*, for dangerous sinks that can all be joined to a complete injectable POP chain. Listing 2 shows an excerpt of a vulnerable application where three *gadgets* are combined to achieve an arbitrary file deletion.

```
1 class File {
2   public function shutdown() {
3     $this->close();
4   }
5   public function close() {
6     fclose($this->h); // harmless
7   }
8 }
9 class TempFile extends File {
10  public function close() {
11    unlink('/var/www/tmp/logs/' . $this->filename); // !!
12  }
13 }
14 class Database {
15  public function __destruct() {
16    $this->handle->shutdown();
17  }
18 }
19 $data = unserialize($_COOKIE['data']);
20 //  O:8:"Database":1:{s:6:"handle";
21 //  O:8:"TempFile":1:{s:8:"filename";s:15:"../../.htaccess";}}
```

Listing 2: Exploitation of a POI vulnerability.

The POI vulnerability occurs in line 19, where user input is deserialized. Note that an application often does not intend to deserialize objects but rather arrays. By forging a cookie with the content seen in lines 20–21, the attacker injects a `Database` object with the `$handle`-property set to a `TempFile` object. Its `$filename` property is then set to the `../../.htaccess` file that the attacker attempts to delete.

When the application terminates, the injected `Database` object will automatically execute its destructor. The destructor will then use the `$handle`-property to execute its `shutdown()` function. Because the attacker loaded the class `TempFile` into this property, the function `shutdown()` of `TempFile` is triggered. It inherits this method from the `File` class. Next, the method `shutdown()` invokes the method `close()`. Although this method is harmless in the `File` class, it is overwritten in the class `TempFile` with a harmful method that deletes the specified `.htaccess` file.

Note that an *initial gadget* (in this case `Database`'s destructor) is required in order to start having an execution flow of already existing code, defined in the object's methods. For every set of objects, multiple variations of *gadgets* can be combined, each leading to another class of vulnerability in the end. As manually checking the application's source for useful *gadgets* is cumbersome and time consuming, an automated approach is needed.

3. STATIC POP CHAIN DETECTION

In order to detect POI vulnerabilities and POP gadget chains in modern applications, an efficient analysis of *object-oriented code* is required—a feature missing in existing analysis frameworks [1,6,7,18,44,46]. In the following, we introduce our approach to address this challenge. In Section 3.1, we first provide a general overview of the taint analysis approach. Afterwards, we review the basic analysis tasks and data flow analysis of procedural PHP code we utilize. We introduce our novel analysis approach for inter-procedural, field-sensitive and object-sensitive data flow analysis in Section 3.3. Based upon this method, we can analyze OOP code for POI vulnerabilities and generate POP gadget chains efficiently (see Section 3.5). We highlight difficulties in the analysis of OOP code and limitations of our approach throughout the section and present small code samples for better understanding.

3.1 High-Level Overview of Taint Analysis

As a basic analysis task, we need to perform a taint analysis capable of inspecting a large number of sensitive sinks, affected parameters, and unsanitized sources. Furthermore, we must precisely analyze built-in functions in order to simulate their actions regarding data flow and data sanitization. To this end, we leverage ideas of static code analysis frameworks based on *block*, *function*, and *file summaries* [6, 46] and extend them to our needs.

When analyzing a given application, each PHP file of the application is transformed into an *Abstract Syntax Tree* (AST). In an initial phase, each AST is analyzed for declarations of functions and classes. The sub trees of these units are extracted from the AST and the remaining tree is assigned to each file's name. Next, each file's AST is transformed into a *Control Flow Graph* (CFG). During this transformation, the AST is split into linked *basic blocks* that represent the control flow of the program. Whenever a new basic block is connected to the current basic block, the current basic block is *simulated*. During this process, the data flow of the current basic block is inferred from its AST, by using an abstract representation of data (details are explained in Section 3.2). The result of the data flow analysis is stored in a *block summary*.

If a sensitive sink occurs during block simulation, the affected argument is analyzed with backwards-directed taint analysis [6, 44]. Similar to our data flow analysis, the origin of the argument is recursively retrieved from the summary of previously connected blocks. If it originates from unsanitized user input, we report a new vulnerability according to the sink's type.

If a user-defined function is called within a basic block, the current simulation is halted and the AST of the called function is transformed into a CFG with the same approach. If a taint analysis during this transformation hits a parameter or global variable of the currently analyzed function, the affected parameter or variable is stored in the *function summary*. Once the CFG transformation is completed, all `return` statements are analyzed in a similar way to determine the returned data of the function. The information is added to the function's summary and the simulation of the callee's context is continued. For every further call of the same function, the function summary is used. Global variables are exported and the arguments of sensitive parameters are analyzed context-sensitively. The function summary will play an important role during POP chain generation (see Section 3.5).

The analysis ends when the ASTs of all files are transformed to CFGs. For each file, a *file summary* is created similar to a function summary for cases in which the file is included multiple times. This analysis approach based on summaries is efficient because every code block is analyzed only once. A remaining challenge is to combine it with an analysis of the highly context-sensitive data flow through objects and methods in object-oriented code.

3.2 Data Flow Analysis

We now present our method of summarizing data flow within a basic block [6,46]. Based on a basic block's AST, we analyze all data assignments to memory locations of the form *loc* := <*assigned data*>. Other forms of data assignments are handled as well, but left aside for brevity reasons. The *assigned data* is transformed into the following *data symbols* which are an abstract representation of data and locations:

- VALUE represents a string or an integer value.
- VARIABLE represents a variable `$x` by its name x.
- ARRAYFETCH represents the access of an array `$x[y]` by its name x and the dimension y. Multiple dimensions are possible, such as for example `$x[y][z]`.

Once the assigned data is transformed into data symbols, its memory location is indexed in the block summary for efficient lookups. In procedural PHP code, the assigned location *loc* is either a variable `$x` or an array dimension `$x[y]`. Assigned data to a variable can be indexed in the block summary by the variable's name. Previously assigned data is overwritten. The assignment to an array dimension requires a more complex representation and this problem is solved by the wrapper symbol ARRAYWRITE. It stores assigned data in a tree graph, whereas the tree's *edges* represent the dimensions and the *leaves* represent the assigned data [18].

Dimension and data are both stored as data symbols. The tree structure allows efficient access to the data by providing one or multiple dimension(s) which are compared to the edges. The ArrayWrite symbol acts as a data wrapper and is indexed in the block summary by the array's name. Further assignments to the same index extend its tree.

To summarize not only the data *assignment* but also the data *flow* of one basic block, the interaction between data assignments is evaluated based up on the current block's summary. For this purpose, the name of an assigned data symbol is looked up in the current summaries' index list to see if it can be *resolved* by previous definitions in the same basic block.

A found Variable symbol is simply replaced with the symbol from the summary. An ArrayFetch symbol has to *carry* its array dimension to the resolved symbol. A resolved Variable symbol will turn into an ArrayFetch symbol with the carried dimension. The dimension of a resolved ArrayFetch symbol is *extended* by the carried dimension. In case the resolved symbol is an ArrayWrite, the symbol mapped to the carried dimension is fetched from the tree.

A return value of a user-defined function is resolved from the function summary. As described in the previous section, it summarizes the data flow of the function's basic blocks. The return value is a data symbol. For simplicity, we ignore the fact that a function can return multiple different data symbols.

Based on these basic blocks' summaries, efficient data flow and taint analysis across linked blocks is possible without requiring the re-evaluating of the blocks' operations.

3.3 Our Approach to OOP Analysis

In this section, we introduce our approach for statically analyzing relevant OOP features for POP detection. First, our prototype gathers information about the object-oriented code (see Section 3.3.1). Then, the allocation of objects (see Section 3.3.2) and the access to object properties (see Section 3.3.3) is included into the data flow analysis. A challenge is to maintain the efficient concept of data flow summaries: with OOP, the context of data handling moves away from a single basic block to a pervasive object that is used in different blocks and functions. We approach this challenge by assisting the backwards-directed data flow analysis with a forwards-directed, object-oriented analysis. For this purpose, new data symbols are added. Finally, our inter-procedural analysis for methods utilizes a *class hierarchy* and *method fingerprints* to handle calls in a context-sensitive manner (see Section 3.3.4).

3.3.1 Initialization

During the initial analysis phase, we extract class definitions from the ASTs. They are stored as part of the analysis process. For static classes, we collect predefined properties and class constants that are transformed to data symbols. During data flow analysis, access to this static content is inferred instantly. Furthermore, we build a *class hierarchy* [8,36] based on the inheritance of each class (e. g., `class A extends B`). To answer the questions *who extends whom* and *who is extended by whom*, it is built in both directions. All defined methods are stored in the analysis environment as user-defined functions, but are linked to their originating class. Additionally, we extract type information of parameters whenever possible.

3.3.2 Object-sensitive Analysis

After initialization, our prototype starts to analyze the data flow of basic blocks as described in Section 3.2. We introduce the new data symbol Object whenever a new object is constructed by the keyword `new`. This data symbol is defined by the instantiated class' name and its properties. The properties are represented by a hash map that references a property name to a data symbol. By default, the map of properties in each Object symbol is empty.

When a new object is created, its constructor is analyzed. A constructor is either the `__construct()` method of the class or a method having the instantiated class' name. Our inter-procedural analyses ensures that all data assigned to properties within the constructor is assigned to the new Object symbol. The details are explained in Section 3.3.4.

Object Propagation.

Then, the created object is assigned to its memory location and indexed in the block summary as described in Section 3.2. As shown in the next sections, certain analysis steps require the knowledge of all present objects and their corresponding class. Thus, at the end of the simulation of one basic block, all the indexed Object symbols are propagated to the next basic block into an *object cache* (illustrated in Figure 1, *dotted* arrow). While our prototype is aware of multiple different objects per code path, we assume for simplicity that no cache index collides.

Moreover, we extract type information from type checks (e.g., `$o instanceof MyClass`) to determine missing class information. The class name is updated in the object cache or a *dummy* object is created if no related object is found.

The object cache is extended by each basic block when new objects are invoked and all objects are propagated until the end of the CFG is reached. This way, each basic block has access to previously invoked objects within its CFG. If the CFG belongs to the main code of a file, the lifetime of all objects passes over. At this point, the object cache is emptied and the `__destruct()` method of each different instantiated class is analyzed. The inter-procedural propagation of objects is explained in Section 3.3.4.

Object-sensitive Magic Methods.

Based on the object cache, special operations on locations pointing to objects are detected and the corresponding magic methods are analyzed (refer to Section 2.1). If the built-in functions `var_export()` or `serialize()` reference a memory location that points to an Object symbol, the corresponding magic methods `__set_state()` or `__sleep()` of the object's class are analyzed (if available). Similarly, the `clone` operator invokes analysis of the method `__clone()` and an implicit or explicit typecast to *String* invokes analysis of the method `__toString()`. If an object is used within a dynamic function call, such as `$object()`, the method `__invoke()` of the object's class is analyzed.

3.3.3 Field-sensitive Analysis

With the knowledge about present objects, our prototype can handle the access to properties. We model writes and reads to properties of objects (i.e., `$o->p`) in a similar way to the access of arrays. The challenge is to maintain object-sensitivity [30]. We refer to the accessed object `$obj` as the *receiving* object, or in short, *receiver* [25].

```
1 $text = 'test';
2 $obj = new MyClass;
3 if(...) {
4    $obj->data = $text;
5 }
6 echo $obj->data;
```

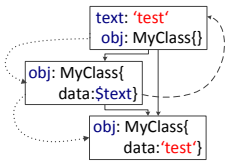

Figure 1: The code on the left creates a new object and assigns data to a property. The corresponding control flow graph is illustrated on the right. The created object `obj` is propagated forward throughout the CFG (*dotted* arrow). Assigned data to an object's property is resolved by backwards-directed data flow analysis (*dashed* arrow).

Property Writes.

A property p of an object `$o` is written to if the location *loc* of the assignment *loc* := <*assigned data*> is a property access (i.e., `$o->p`). We then first try to resolve the *assigned data* by backwards-directed data flow analysis of all previously linked blocks' summaries (recall Section 3.2).

If the receiver's name `$o` is found in the object cache of the current basic block, then the assigned data's symbol is added to the property hash map of object `$o` in the object cache with index p. In case an array dimension of a property is accessed (i.e., `$o->p[d]`), the assigned data is wrapped into an ArrayWrite symbol. An example is given in Figure 1. Here, the variable `$text` is resolved in line 4 and its value *test* is assigned to the object's property.

However, during intra-procedural analysis, the object cache is not always complete. For example, when `$o` is a parameter or a global variable of the current function (see Listing 3), or the receiver's name is the reserved variable `$this` that refers to the current object of the called method, the receiver is unknown. In this case, we use the wrapper symbol PropertyWrite to save the information about the receiver's name, property dimension, and assigned data symbol. All PropertyWrite symbols of one basic block are stored in its *propwrite* cache. This cache is propagated through all upcoming basic blocks, similar to the *object* cache. The details on how the property writes are assigned to the correct receiver during inter-procedural analysis are explained in Section 3.3.4.

Furthermore, we handle writes to *static* properties. Similar to the access of non-static properties, the receiver class can be related to the current callee's class (e. g., `self::$p` or `parent::$p`), or to a secondary class (e.g., `Class::$p`). In both cases, the target class name is determined from the class hierarchy and the assigned data is stored in the prototype's environment for later access.

Property Access.

We introduce the data symbol PropertyFetch to model the *access* of a property. It extends the ArrayFetch symbol with a *property* dimension. This way, a PropertyFetch symbol is also capable of having an array dimension. The name associated to the symbol is the name of the receiving object. For example, the code `$v = $o->p[a]` assigns a PropertyFetch symbol with the name o, the property dimension p, and the array dimension a to the location v. During data flow analysis, we try to *resolve* this symbol.

The PropertyFetch symbol can be resolved from the block summary if the receiver name o is found in the object cache. First, the property dimension p is fetched from the hash map and then the array dimension a is carried to the resolved symbol. If the receiver name o is indexed in the data flow summary, the receiver's symbol is fetched and the object's property dimension p is *carried* to it. In this process, a Variabe symbol is inferred into a PropertyFetch symbol with a property dimension p. An ArrayFetch symbol is inferred similarly, but carries its array dimension to the PropertyFetch symbol. If a PropertyFetch symbol is resolved from the block summary into another PropertyFetch symbol, the property dimensions are added. Finally, if the PropertyFetch symbol was not inferred from the block summary or the object cache, it is looked up in the *propwrite* cache. Otherwise the PropertyFetch symbol remains unresolved.

Field-sensitive Magic Methods.

We also invoke analysis of magic methods for certain operations on PropertyFetch symbols. However, this is only possible when the class name of the receiver is resolved from the object cache. Then, if the built-in function `isset()` or `unset()` references to an inaccessible property (determined by the class definition), the magic method `__isset()` or `__unset()` of the receiver's class is analyzed. Furthermore, if the property dimension of a property read or write is not defined in the receiver's class, the magic method `__get()` or `__set()` is analyzed. When the receiver's class name cannot be resolved, no further analysis invoked. Note that in case of a POI vulnerability, an object of an arbitrary class is present so that field-sensitive magic methods are still supported for POP chain generation by considering all available classes (for details refer to Section 3.5).

3.3.4 *Inter-procedural Analysis*

Our prototype handles calls to methods in a way similar to user-defined functions. However, because a method name can be defined in multiple classes, our prototype has to determine the receiver's class to invoke the analysis of the correct method [30].

Challenge: Receiver Analysis.

A call to a static method is easily mapped to the correct class by its specified name (e. g., `Class::method()`). In case the static keywords `self::method()` or `parent::method()` are used, the class name can be resolved from the class hierarchy of the current method's class [8,36]. The same applies if the reserved variable `$this` is used as receiver.

For all other non-static method calls, such as `$o->method()`, the class name has to be inferred from the receiver variable `$o`. If the receiver's name is found in the current block's object cache, the class name is extracted from the cached Object symbol. Note that the object cache contains only objects that were created in the current CFG or imported into the current CFG as return value of a function. However, as shown in Listing 3, if the receiver is passed as an argument (`$obj1`) or global variable (`$obj2`) to the currently analyzed method, no information about the receiver is available. The callee's context is only applied to the function summary, while our intra-procedural analysis is context-insensitive.

```
1 public function handler($obj1) {
2    $obj1->method1(1, 2);
3    global $obj2;
4    $obj2->method2(1, 2, 3);
5 }
```

Listing 3: Receiver `$obj1` and `$obj2` are unknown.

We approach the problem for `$obj1` by searching for all available methods named `method1()` in all class definitions. If the name is unique, the corresponding method is invoked. Otherwise, we compare the number of arguments (here: two) to the number of parameters specified in the method declarations. Then, we invoke the analysis for all matching candidates and combine their function summaries to one summary. While this approach can potentially lead to an over-approximation, it is likely that methods, such as the method `handler()` in Listing 3, are intended to call different methods on different objects.

For `$obj2` we take a different approach. In our initial setup phase, we index the name of all global variables within all application's functions and methods identified by the `global` keyword or `$GLOBALS` variable. If a new object is assigned to a location having one of these indexed names, the object's class name is referenced to the index. During intra-procedural analysis, the class name can then be retrieved for global variables. In case of *dynamic* global variables we fall back to the approach as described for `$obj1`.

Invocation-sensitive Magic Methods.

For static method calls we check the accessibility of the method regarding to the receiver's class name with the help of our class hierarchy. We invoke any defined `__static-Call()` method of that class if the method is not accessible. The same applies to the `__call()` method for non-static method calls. Similar to the analysis of field-sensitive magic methods, our approach is limited by the success of our receiver analysis. However, during object injection, all classes are considered so that our analysis of invocation-sensitive magic methods for gadget chain generation is not limited.

Context-Sensitivity.

Once the correct method is identified and its CFG analysis is completed, post conditions of the method call are applied to the callee's context. In a way similar to functions, the summary of a method provides return values, sensitive parameters, and sensitive global variables. If a taint analysis of a sensitive sink within a method results in an unresolved PropertyFetch symbol and the receiver's name is either `$this`, a parameter's name, or a global variable's name, the symbol is added to the function summary as *sensitive property*. When the method is called, the sensitive parameters, sensitive globals, and sensitive properties are adjusted to the callee's arguments and a new taint analysis is invoked from the callee's context. Furthermore, the *object* and *propwrite* cache is propagated from the function summary to the callee's basic block. However, objects are only propagated if their receiver is a global variable or a return value of the method. Other objects are deleted from the cache and their destructor is invoked.

Property writes are applied to global receivers as well as to receivers that were passed by parameter. The receiver name is adjusted to the arguments of the method call. Property writes to the receiver `$this` are applied to the receiver of the method call.

3.4 *Case Study:* POI Detection in Contao CMS

We now discuss a POI vulnerability in Contao CMS to demonstrate the complexity of real-world OOP code and to illustrate our novel approach of analyzing OOP code. The affected code of Contao CMS is given in Listings 4–6.

```
1  class PagePicker extends Backend {
2     public function run() {
3        if ($_POST && Environment::get('isAjaxRequest')) {
4           $this->objAjax = new Ajax(Input::post('action'));
5        }
6        ...
7        if ($_POST && Environment::get('isAjaxRequest')) {
8           $this->objAjax->execPostActions($objDca);
9        }
10    }
11 }
12 $objPagePicker = new PagePicker();
13 $objPagePicker->run();
```

Listing 4: The method `run()` of the class `PagePicker`.

Our analysis begins in line 12 of Listing 4, where a new Object symbol is created and indexed in the block summary under the name *objPagePicker*. We neglect the constructor analysis. In the next line, the method `run()` is called. Its class is determined from the recently indexed Object symbol. Our analysis continues intra-procedurally in the first basic block of the method `run()` in line 4. Here, a new object of the class `Ajax` is instantiated and assigned to the property `$this->objAjax`. Again, we omit the constructor analysis. The receiver `$this` is unknown at that time. Thus, we store the new object into a PropWrite symbol. It assigns the Object symbol `Ajax` to the property `objAjax` of the receiver `this`. The PropWrite symbol is stored in the *propwrite* cache and propagated to each further basic block within the method `run()`. Consequently, in line 8, the receiver `$this->objAjax` of the call `execPostActions()` is resolved to the `Ajax` object from the *propwrite* cache. After this call, the analysis of `run()` terminates and the property write to `objAjax` is applied to the receiver `$objPagePicker`.

```
1  class Ajax extends Backend {
2     public function execPostActions(DataContainer $dc) {
3        if ($dc instanceof DC_Table) {
4           echo $dc->editAll($this->ajaxId, $id);
5        }
```

Listing 5: The method `execPostActions()` of the class `Ajax`.

In Listing 5, the executed method `execPostActions()` is shown. Due to our context-insensitive intra-procedural analysis, arguments passed to a method are unknown during analysis time. Thus, the receiver `$dc` of the call `editAll()` in line 4 is unknown. However, our prototype is able to infer the class information from the parameter specification (*DataContainer*) and more specifically from the `if`-condition in line 3 (*DC_Table*). Otherwise, the correct method would have been found by *method fingerprinting*. There are two methods defined with the name `editAll()`, but only one accepts two parameters by its specification.

```
1  class DC_Table {
2     public function editAll($intId=null, $ajaxId=null) {
3        if (Input::post('FORM_SUBMIT')) {
4           $session = unserialize(Input::post('all_fields'));
5        }
```

Listing 6: The method `editAll()` of the class `DC_Table`.

Listing 6 shows the called method `editAll()` of the class `DC_Table`. It contains the actual POI vulnerability in line 4. Here, user input is fetched from the static class `Input` and is passed to the sensitive function `unserialize()`.

3.5 POP Chain Generation

Whenever our analysis reports a call to `unserialize()` as vulnerable, the return value of the `unserialize()` call is an OBJECT symbol with a special *POI* flag set to *true*. If the return value of this `unserialize()` call is assigned to a variable, the flagged *Object* symbol is added to the current block's object cache that is propagated through the upcoming basic blocks, as described in the previous section. However, its flag causes certain different analysis steps regarding calls to magic methods.

First, all `__wakeup()` methods of all classes are analyzed as *initial gadgets*. If an object-sensitive magic method is invoked on a flagged OBJECT symbol, all magic methods of its type are also analyzed. This applies as well to a field-sensitive or invocation-sensitive magic method that is invoked on a flagged OBJECT symbol as receiver. The inter-procedural analysis of the magic methods is performed with an important difference: All sensitive properties of the function summary immediately report a POP gadget chain because the attacker has control over the object's properties.

Furthermore, we limited gadget chains to only severe vulnerabilities by deactivating the detection of client-side vulnerabilities, such as cross-site scripting and open redirects, in our prototype implementation. We also omit vulnerabilities that are triggered by a context-independent magic method and cannot be exploited, such as path traversal attacks against file handlers without further processing. An exemplary POP analysis and report is presented in Section 3.6.

Our approach has two remaining challenges. Recall Listing 3 where an object is unknown at intra-procedural analysis time. If we assume that `method1()` or `method2()` is a magic method, we do not know at the time of the intra-procedural analysis if the object is flagged or not. Thus, we do not know if all magic methods should be analyzed or not. We approach this problem by setting a different flag for each invoked magic method on an unknown receiver in the function summary. When a method is called with a flagged object as argument, we can tell from the function summary during inter-procedural analysis which magic method was invoked and we trigger its analysis.

A false gadget chain report occurs if a magic method of a class that is shipped with the project is analyzed, although the class is not loaded at runtime within the executed code path. We approach this problem by creating a stack of included files [14] during analysis *on-the-fly*. Before a magic method is analyzed, the file name of the method's class is confirmed in the stack in order to prove its availability. This routine is ignored if a class autoloader is detected [38].

3.6 *Case Study:* POP Chain in Contao CMS

We now introduce a previously unreported gadget chain in Contao CMS leading to an arbitrary file delete vulnerability. The chain is invoked through the `__destruct()` method of the class `Swift_Mime_SimpleEntity` that is available through an autoloader. This initial gadget is shown in Listing 7 and it is automatically analyzed, when the flagged OBJECT symbol of a POI is removed from the object cache. In line 3, our prototype invokes the analysis of all available `clearAll()` methods within the application's code base because the receiver `$this->_cache` is unknown. It can be arbitrarily specified during object injection and point to any `clearAll()` method.

```
1 class Swift_Mime_SimpleEntity {
2   public function __destruct() {
3     $this->_cache->clearAll();
4   }
```

Listing 7: Initial POP gadget in Contao CMS.

There are four `clearAll()` methods available in the code base. While three of them are harmless, the one in the class `Swift_KeyCache_Disk` triggers another gadget. As shown in Listing 8, in line 3, it calls the function `clearKey()`. The receiver of this call is the reserved variable `$this`. Thus, only methods within the same class or its class hierarchy are considered and the method defined in line 5 is the only candidate.

```
1 class Swift_KeyCache_Disk {
2   public function clearAll()
3     $this->clearKey();
4   }
5   public function clearKey()
6     unlink($this->_path);
7   }
```

Listing 8: Final POP gadget leading to arbitrary file delete.

Here, the property `_path` is used in the sensitive built-in function `unlink()` that deletes a file. Our prototype transfers the sensitive property `_path` to the receiver `$this->_cache` in the `__destruct()` method, where it issues a vulnerability report as shown in Listing 9. The POP chain report is then attached to the POI vulnerability report.

```
Unserialize() to File Delete (unlink)
 Swift_Mime_SimpleEntity::__destruct()
 Swift_Mime_SimpleEntity->_cache = Swift_KeyCache_Disk
 unlink(Swift_KeyCache_Disk->_path)
```

Listing 9: Generated POP chain report of our prototype.

4. EVALUATION

We implemented a prototype of the approach introduced in the previous section as an extension of our static code analysis framework RIPS [6]. To evaluate its effectiveness, we examined the CVE database regarding PHP object injection vulnerabilities in modern PHP applications [26]. Out of the CVE entries published in the years 2013 and 2014, we chose applications according to the following criteria:

- The vulnerable software version is still available for download so that we can replicate the vulnerability.
- The application is non-trivial (i.e., has more than 40K LOC) and is primarily written in object-oriented code.
- The affected application is exploitable *as it is*. For example, we excluded third-party plugins or framework components that require an implementation.

We selected nine CVE entries matching our criteria and also included Piwik as the first reported software with a POI vulnerability. The list of selected applications is given in Table 1. Our selection includes some of the most popular PHP applications on the Web [42].

Approximately, each of our selected application consists of 700 PHP files and about 170K lines of PHP code (LOC). The analysis was performed on a machine with an Intel i7-2600 CPU @ 3.40 GHz and 16 GB RAM. On average, our

Table 1: Evaluation results for selected applications recently affected by a POI vulnerability. The number of POI vulnerabilities and chains detected by our prototype are compared to the number of previously known issues. **Highlighted** numbers indicate cases were our prototype detected novel POI vulnerabilites or POP chains.

CVE Number	Software	Version	Files	LOC	Time [s]	Mem [MB]	POI	Gadgets	Chains
CVE-2014-2294	Open Web Analytics	1.5.6	463	82 013	155	475	0/1	24	**9/0**
CVE-2014-1860	Contao CMS	3.2.4	578	202 993	298	1 264	**19/3**	136	**14/3**
CVE-2014-0334	CMS Made Simple	1.11.9	692	135 478	567	922	1/1	41	**1/0**
CVE-2013-7034	LiveZilla	5.1.2.0	103	42 753	151	342	**2/1**	21	0/0
CVE-2013-4338	Wordpress	3.5.1	425	190 800	1 138	7 640	0/1	41	0/0
CVE-2013-3528	Vanilla Forums	2.0.18.5	597	123 465	951	6 471	2/2	14	0/1
CVE-2013-2225	GLPI	0.83.9	1 025	347 682	676	1 632	**15/1**	77	0/0
CVE-2013-1465	CubeCart	5.2.0	846	141 404	447	1 483	1/1	47	**3/1**
CVE-2013-1453	Joomla	3.0.2	1 592	289 207	338	1 251	**2/1**	73	**5/2**
CVE-2009-4137	Piwik	0.4.5	750	174 314	87	476	1/1	111	**4/3**
Total			7 071	1 730 109	4 808	21 956	**43/13**	585	**36/10**

prototype implementation required 8 minutes and about 2 GB of memory to perform the POI and POP analysis for a given application. We believe that our efficient concept of using block and function summaries also applies to larger code bases and that our results clearly outperform manual code analysis.

In the following, we present an evaluation of the reported PHP object injection vulnerabilities for each application (see Section 4.1). Then, we study how many gadgets are available in each application (see Section 4.2) and how many gadget chains our prototype was able to connect to a new vulnerability (see Section 4.3).

In total, we were able to find 30 new vulnerabilities and 28 previously undocumented chains. Overall, our evaluation results show that two POI vulnerabilities and two known chains were missed by our current prototype implementation. Furthermore, false positives occurred only during the chain detection in one application. We also discuss the reasons for these false negatives and false positives throughout this section.

4.1 POI Detection in OOP Code

As a first step, we verified if our prototype detects the POI vulnerabilities described in the CVE entries. We compare the number of reported POI vulnerabilities by our prototype to the number of described vulnerabilities in each CVE in the column *POI* of Table 1. For 8 out of 10 vulnerable applications, at least one POI was detected. For four applications, our prototype even found at least one novel POI vulnerabilities that is not included in the CVE. We believe that these vulnerabilities were missed during manual analysis. Our prototype reported no false POI vulnerabilities.

The novel POI vulnerabilities are fixed in the latest LiveZilla 5.2.0.1, Contao CMS 3.2.9 and GLPI 0.84.5 by replacing calls to `unserialize()` with `json_decode()`, or by sanitizing user input. However, the POI in CMS Made Simple was not fixed in the latest release yet, because no chain was found. Our prototype detected a novel gadget chain to delete arbitrary files and we reported the issue to the developers. Our novel POI in Joomla also exists in the latest version 3.3.0 and we reported the issue as well.

The POI vulnerability in Open Web Analytics and Wordpress was *not* detected by our prototype. The root cause for the false negative in Open Web Analytics is the insufficient analysis of reflection, which is an unsolved problem in the field of static analysis [4, 14, 22].

```
1 class owa_coreAPI {
2   public static function classFactory($module, $class) {
3       return owa_lib::factory(OWA_BASE_DIR.'/modules/'.
            $module.'/classes/', $class);
4   }
5   public static function getRequestParam($name) {
6       $service = owa_coreAPI::classFactory('base', 'service')
            ;
7       return $service->request->getParam($name);
8   }
```

Listing 10: Dynamic class factory in Open Web Analytics.

The simplified code is shown in Listing 10. In Open Web Analytics, every access to user input is performed via the static method `getRequestParam()` defined in line 5. This method fetches a new object through the method `classFactory()` in line 6 and calls the method `getParam()` on the `request` property as receiver. Because the method `factory()` used in `classFactory()` internally uses reflection, no knowledge about the object assigned to `$service` in line 6 is available to our prototype. The prototype can still fingerprint the method `getParam()`, but this method accesses properties of the object assigned to the property `request`. Its properties are filled during the dynamic object construction in the factory. We plan to improve the analysis of dynamic OOP code in the future.

The false negative in Wordpress is based on second-order data flow [7]: metadata about a user is stored in a database and later loaded into a cache before it is deserialized. The database queries are constructed dynamically and cannot be reconstructed completely by our prototype in order to recognize the data flow.

4.2 Available POP Gadgets

We let our prototype report all declared non-empty magic methods in our selected applications to establish a ground truth. On average, there are about 59 potential initial gadgets available per application. The different amounts of magic methods are listed in Table 2. In our evaluation, the most common magic methods are `__set()` and `__get()` methods. However, since they implement the simple logic for missing getter and setter methods, none of them was exploitable. Among the available gadgets, the `__destruct()` method is also frequently present. It provides the best chance for abusable code because it is context-independent. The context-dependent method `__toString()` is defined often, but is supposed to return a string representation of the object which does not yield a high chance of abusable PHP

Table 2: Gadget distribution within our selected applications. **Highlighted** numbers indicate initial gadgets of chains.

Software	set	get	toString	destruct	isset	call	wakeup	unset	clone	set_state	callStatic	total
Contao CMS	47	32	**16**	**17**	12	3	4	2	2	0	1	136
Piwik	11	19	**23**	**21**	9	9	**8**	8	3	0	0	111
GLPI	43	23	5	1	1	0	0	0	0	4	0	77
Joomla	4	11	30	**15**	1	4	4	1	3	0	0	73
CubeCart	8	11	4	**18**	1	3	0	1	1	0	0	47
Wordpress	4	6	13	8	5	2	0	2	0	0	1	41
CMS Simple	8	15	7	**3**	2	3	1	0	2	0	0	41
OWA	2	2	3	**15**	2	0	0	0	0	0	0	24
LiveZilla	1	6	4	5	1	3	0	1	0	0	0	21
Vanilla	3	3	4	1	0	3	0	0	0	0	0	14
Total	131	128	109	104	34	30	17	15	11	4	2	585

Table 3: Distribution of different vulnerability types in our detected POP gadget chains.

Software	FD	FC	FM	SQLi	LFI	XXE
OWA	2	2	1	3	1	-
Contao	6	5	3	-	-	-
CMS Simple	1	-	-	-	-	-
CubeCart	1	-	-	2	-	-
Joomla	1	-	2	1	1	-
Piwik	-	2	1	-	-	1
Total	11	9	7	6	2	1

code. The least frequent magic methods are `callStatic()` and `__invoke()` which we did not find in any of our selected applications. Based on the low number of gadgets, we expected the POI in Vanilla Forums (14), LiveZilla (21), and Open Web Analytics (24) to be less likely exploitable compared to, for example, Contao CMS (136) or Piwik (111).

Note that the number of available gadgets does not significantly influence the overall performance. That is, first of all, due to the fact that the code size of magic methods is often rather small. Furthermore, some of them are already included in our normal OOP analysis. Second, all further gadgets in a chain are user-defined methods. These are analyzed for POI vulnerabilities by our prototype. Because the analysis results are stored in the method's summary, they can be re-used when building a chain with little effort.

4.3 Detected POP Gadget Chains

Next, we evaluated the reported POP gadget chains of our prototype. For Wordpress and Open Web Analytics, we simplified the POI vulnerability so that our prototype was capable of detecting the vulnerability after which we can include the applications in our gadget chain evaluation.

The total number of exploitable gadget chains reported by our prototype is compared to the known gadget chains from security advisories in the column *Chains* in Table 1. In total, 36 exploitable gadget chains were reported. Our prototype successfully detected a gadget chain in 6 out of 10 applications, whereas 28 gadget chains were previously unknown. Starting from the initial gadget to the sensitive sink, the length of detected gadget chains ranges from 1 up to 8 gadgets with an average chain length of 3 gadgets. Table 2 highlights the magic methods used as an *initial gadget* with a bold number. The most abused magic method was `__destruct()`, used by 86% of the gadget chains. Only four gadget chains initially exploited `__toString()` and one chain exploited `__wakeup()`.

The number of different vulnerability types detected in each application through POP is listed in Table 3. The most prominent vulnerability types are file delete (FD), file create (FC), and file modification (FM) vulnerabilities. Furthermore, SQL injection (SQLi) and local file inclusion (LFI) vulnerabilities were detected, as well as one XML external entity injection (XXE).

Surprisingly, 9 chains were found in Open Web Analytics, although only 24 initial gadgets are available. However, one call to a method with a frequently used name is enough to jump to a large portion of the application's code. Due to one dynamic class invocation (refer to Section 4.1) also 10 false positives occurred. For LiveZilla, Wordpress, and GLPI, no gadget chain was detected by our tool. However, since no gadget chain is publicly documented, we assume that the POI vulnerability is not exploitable with the application's core code. A false negative occurred in Piwik and in Vanilla Forums. Here, our prototype analyzed dynamic OOP features imprecisely.

5. RELATED WORK

Code reuse attacks and OOP analysis were extensively studied over the last years. However, both techniques have not been applied to PHP-based web applications before. In this section, we review related work of both fields.

5.1 Code Reuse Techniques

The idea of reusing existing code instead of injecting shellcode goes back to Solar Designer, who was the first to publicly documented such an attack to bypass a non-executable stack [32]. The idea is that the adversary constructs a fake call stack which contains the necessary parameters and meta-information (i.e., a return address that points to a library function). After a successful exploit, the vulnerable function attempts to return, but the fake call stack leads to a diversion of the control flow. While an adversary can return to an arbitrary location, she typically returns to one of the functions provided by the C standard library and thus such attacks are called *return-to-libc*.

This basic idea was extended in the following years, leading to the technique nowadays referred to as *return-oriented programming* (ROP) [21, 27]. Instead of reusing complete functions, an adversary can also chain small code fragments and build a malicious payload. There are ROP compilers [16, 29] capable of automatically converting a given piece of code into an application-specific ROP chain and Snow et al. demonstrated how such chains can be built on-the-fly [31]. Another technique closely related to ROP leverages

gadgets not ending in return instructions but some kind of indirect jumps [3, 5].

Back in 2009/2010, Esser gave two presentation in which he described the idea of applying code reuse attacks in the context of PHP-based web applications [9, 10]. He demonstrated the practical feasibility of such an approach and coined the term *property-oriented programming* (POP). We build upon this work and propose a static analysis approach to detect PHP object injection vulnerabilities and POP gadget chains in an automated way.

5.2 Analysis of Web Applications

Due to the practical importance of PHP-based web applications, a large number of techniques to analyze such applications for potential (injection) vulnerabilities were developed (e.g., [1, 6, 7, 15, 18, 20, 33, 44–46]). Our static code analysis is based on block and function summaries, a concept first introduced by Xie and Aiken [46]. Previously, we extended it for precise analysis of PHP built-in features [6] and for second-order vulnerability detection [7]. In this work, we extended our procedural data flow analysis with support for relevant object-oriented features for POI detection. To the best of our knowledge, none of the existing approaches is able to analyze object-oriented PHP code.

The challenges we address to perform an efficient OOP analysis on large applications is a research topic addressed for other kinds of programming languages. A broad overview on different approaches to perform object-sensitivity analysis was performed by Smaragdakis et al. [30]. They introduce type-sensitive analysis as a more scalable solution that picks its context based on types instead of objects. Although their approach looks promising, it is not applicable to a weakly-typed language such as PHP.

Several static code analysis approaches have been proposed to perform points-to analysis for the weakly-typed JavaScript language [2, 11, 12, 17, 35]. Similar work was also performed for the Java language [23, 25, 40]. For example, Livshits and Lam proposed a static analysis approach to detect security vulnerabilities in Java applications [23]. Tripp et al. designed static taint analysis for Java and implemented their approach in the TAJ system [40]. In general, these approaches cannot be adopted to the PHP language due to missing type information in PHP.

6. CONCLUSION AND FUTURE WORK

Code reuse attacks are not only a threat for memory corruption vulnerabilities in binary executables, but also for the web application domain. In this paper, we studied the nature of PHP object injection vulnerabilities that can be exploited via property-oriented programming. In such code reuse attacks, an object with modified properties is injected into the application. Through PHP's magic methods, the control flow is diverted and an adversary can perform malicious computations. We proposed and implemented an automated approach for efficient gadget chain detection. An empirical evaluation demonstrates that our method can find new POI vulnerabilities and different kinds of gadget chains.

Our prototype models only relevant OOP features for POP detection. False positives and negatives can occur by imprecise handling of dynamic OOP features [14]. Future work will extend the support of OOP features and address the challenge of framework analysis [24, 34].

7. REFERENCES

[1] Balzarotti, D., Cova, M., Felmetsger, V., Jovanovic, N., Kirda, E., Kruegel, C., and Vigna, G. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *IEEE Symposium on Security and Privacy* (2008).

[2] Barth, A., Weinberger, J., and Song, D. Cross-Origin JavaScript Capability Leaks: Detection, Exploitation, and Defense. In *USENIX Security Symposium* (2009), pp. 187–198.

[3] Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z. Jump-oriented Programming: A New Class of Code-reuse Attack. In *ACM Symposium on Information, Computer and Communications Security (ASIACCS)* (2011).

[4] Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., and Mezini, M. Taming Reflection: Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ICSE '11, pp. 241–250.

[5] Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. Return-oriented programming without returns. In *ACM Conference on Computer and Communications Security (CCS)* (2010).

[6] Dahse, J., and Holz, T. Simulation of Built-in PHP Features for Precise Static Code Analysis. In *Symposium on Network and Distributed System Security (NDSS)* (2014).

[7] Dahse, J., and Holz, T. Static Detection of Second-Order Vulnerabilities in Web Applications. In *USENIX Security Symposium* (2014).

[8] Dean, J., Grove, D., and Chambers, C. Optimization of Object-oriented Programs using Static Class Hierarchy Analysis. In *ECOOP'95 Object-Oriented Programming, 9th European Conference, Åarhus, Denmark, August 7–11, 1995* (1995), Springer, pp. 77–101.

[9] Esser, S. Shocking News in PHP Exploitation. In *Power of Community (POC)* (2009).

[10] Esser, S. Utilizing Code Reuse Or Return Oriented Programming in PHP Applications. In *BlackHat USA* (2010).

[11] Guarnieri, S., and Livshits, V. B. GATEKEEPER: Mostly Static Enforcement of Security and Reliability Policies for JavaScript Code. In *USENIX Security Symposium* (2009), pp. 151–168.

[12] Guarnieri, S., Pistoia, M., Tripp, O., Dolby, J., Teilhet, S., and Berg, R. Saving the World Wide Web from Vulnerable JavaScript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (2011), ACM, pp. 177–187.

[13] Halfond, W. G., Viegas, J., and Orso, A. A Classification of SQL Injection Attacks and Countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering* (2006).

[14] Hills, M., Klint, P., and Vinju, J. An Empirical Study of PHP Feature Usage. In *International Symposium on Software Testing and Analysis (ISSTA)* (2013).

[15] Huang, Y.-W., Yu, F., Hang, C., Tsai, C.-H., Lee, D.-T., and Kuo, S.-Y. Securing Web Application Code by Static Analysis and Runtime Protection. In *International Conference on the World Wide Web (WWW)* (2004).

[16] Hund, R., Holz, T., and Freiling, F. C. Return-oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms. In *USENIX Security Symposium* (2009).

[17] Jang, D., and Choe, K.-M. Points-to analysis for JavaScript. In *Proceedings of the 2009 ACM Symposium on Applied Computing* (2009), ACM, pp. 1930–1937.

[18] Jovanovic, N., Kruegel, C., and Kirda, E. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities (Short Paper). In *IEEE Symposium on Security and Privacy* (2006).

[19] Klein, A. Cross-Site Scripting Explained. *Sanctum White Paper* (2002).

[20] Kneuss, E., Suter, P., and Kuncak, V. Phantm: PHP Analyzer for Type Mismatch. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)* (2010).

[21] Krahmer, S. x86-64 Buffer Overflow Exploits and the Borrowed Code Chunks Exploitation Technique. `http://users.suse.com/~krahmer/no-nx.pdf`, 2005.

[22] Livshits, B., Whaley, J., and Lam, M. S. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (2005), APLAS'05, pp. 139–160.

[23] Livshits, V. B., and Lam, M. S. Finding Security Vulnerabilities in Java Applications with Static Analysis. In *USENIX Security Symposium* (2005).

[24] Madsen, M., Livshits, B., and Fanning, M. Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (2013), ESEC/FSE 2013, ACM, pp. 499–509.

[25] Milanova, A., Rountev, A., and Ryder, B. G. Parameterized Object Sensitivity for Points-to and Side-effect Analyses for Java. *ACM SIGSOFT Software Engineering Notes 27*, 4 (2002), 1–11.

[26] MITRE. Common Vulnerabilities and Exposures (CVE). `http://cve.mitre.org/`, as of May 2014.

[27] Roemer, R., Buchanan, E., Shacham, H., and Savage, S. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security 15*, 1 (Mar. 2012).

[28] Schwartz, E. J., Avgerinos, T., and Brumley, D. All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). In *IEEE Symposium on Security and Privacy* (2010).

[29] Schwartz, E. J., Avgerinos, T., and Brumley, D. Q: Exploit Hardening Made Easy. In *USENIX Security Symposium* (2011).

[30] Smaragdakis, Y., Bravenboer, M., and Lhoták, O. Pick Your Contexts Well: Understanding Object-sensitivity. *ACM SIGPLAN Notices 46*, 1 (2011), 17–30.

[31] Snow, K. Z., Monrose, F., Davi, L., Dmitrienko, A., Liebchen, C., and Sadeghi, A.-R. Just-in-Time Code Reuse: On the Effectiveness of Fine-grained Address Space Layout Randomization. In *IEEE Symposium on Security and Privacy* (2013).

[32] Solar Designer. Getting around non-executable stack (and fix). `http://seclists.org/bugtraq/1997/Aug/63`, as of May 2014.

[33] Son, S., and Shmatikov, V. SAFERPHP: Finding Semantic Vulnerabilities in PHP Applications. In *ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)* (2011).

[34] Sridharan, M., Artzi, S., Pistoia, M., Guarnieri, S., Tripp, O., and Berg, R. F4F: Taint Analysis of Framework-based Web Applications. *ACM SIGPLAN Notices 46*, 10 (2011), 1053–1068.

[35] Sridharan, M., Dolby, J., Chandra, S., Schäfer, M., and Tip, F. Correlation Tracking for Points-to Analysis of JavaScript. In *ECOOP 2012–Object-Oriented Programming*. Springer, 2012, pp. 435–458.

[36] Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., and Godin, C. Practical Virtual Method Call Resolution for Java. *ACM SIGPLAN Notices 35*, 10 (2000), 264–280.

[37] Szekeres, L., Payer, M., Wei, T., and Song, D. SoK: Eternal War in Memory. In *IEEE Symposium on Security and Privacy* (2013).

[38] The PHP Group. PHP: Autoloading Classes. `http://php.net/manual/language.oop5.autoload.php`, as of May 2014.

[39] The PHP Group. PHP: Magic Methods. `http://php.net/manual/language.oop5.magic.php`, as of May 2014.

[40] Tripp, O., Pistoia, M., Fink, S. J., Sridharan, M., and Weisman, O. TAJ: Effective Taint Analysis of Web Applications. *ACM Sigplan Notices 44*, 6 (2009), 87–97.

[41] van der Veen, V., Dutt-Sharma, N., Cavallaro, L., and Bos, H. Memory Errors: The Past, the Present, and the Future. In *Symposium on Recent Advances in Intrusion Detection (RAID)* (2012).

[42] W3Techs. Usage of Content Management Systems for Websites. `http://w3techs.com/technologies/overview/content_management/all`, as of May 2014.

[43] W3Techs. Usage of Server-side Programming Languages for Websites. `http://w3techs.com/technologies/overview/programming_language/all`, as of May 2014.

[44] Wasserman, G., and Su, Z. Static Detection of Cross-Site Scripting Vulnerabilities. In *International Conference on Software Engineering (ICSE)* (2008).

[45] Wassermann, G., and Su, Z. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2007).

[46] Xie, Y., and Aiken, A. Static Detection of Security Vulnerabilities in Scripting Languages. In *USENIX Security Symposium* (2006).