QUACK: Hindering Deserialization Attacks via Static Duck Typing

Yaniv David*, Neophytos Christou†, Andreas D. Kellas*, Vasileios P. Kemerlis†, and Junfeng Yang*
*Columbia University †Brown University

***Abstract*—Managed languages facilitate convenient ways for serializing objects, allowing applications to persist and transfer them easily, yet this feature opens them up to attacks. By manipulating serialized objects, attackers can trigger a chained execution of existing code segments, using them as gadgets to form an exploit. Protecting deserialization calls against attacks is cumbersome and tedious, leading to many developers avoiding deploying defenses properly. We present QUACK, a framework for automatically protecting applications by fixing calls to deserialization APIs. These fixes limit the classes allowed for usage in the deserialization process, severely limiting the code available for use as part of exploits. QUACK calculates the set of classes that should be allowed using a novel static duck typing inference technique. QUACK statically collects all statements in the code that manipulate the object after it is deserialized, and uses the evidence in these statements to filter a list of classes that must be available into the set of classes used in practice. We have implemented QUACK for PHP and evaluated it on a set of applications with known CVEs and popular applications crawled randomly from GitHub. QUACK managed to fix the applications in a way that prevented any attempt at automatically generating an exploit against them by blocking, on average, 97% of the application's code that could be used as gadgets. We submitted a sample of three fixes generated by QUACK as pull requests and their developers already merged them.**

I. INTRODUCTION

A hallmark of most managed programming languages is their memory safety: the language runtime tracks object sizes and lifetimes to ensure that objects are always accessed within bounds (e.g., no off-by-one errors) and live (e.g., no use-after-free errors). Besides eliminating memory errors, another major benefit is that the runtime provides developers with APIs that automatically serialize (and deserialize) an object into (and from) a low-level representation. Deserialization is extremely useful because it decouples the object's lifetime from the program execution. Developers can use it to persist objects—including all their transitive fields—to files or databases, or to transfer objects between nodes for distributed or heterogeneous computing (e.g., storing machine-learning models [54]), all without writing any (de)serialization code. A simple search on GitHub reveals more than 5.4M PHP [17] and 10M Python invocations of their respective deserialization APIs.

By design, the class information of serialized objects is stored within the serialized representation itself. This can introduce vulnerabilities to applications that deserialize *untrusted* input, since attackers can provide *malicious* input that creates objects of *arbitrary* classes of their choice when deserialized. Attackers have devised various exploitation techniques that weaponize these vulnerabilities. For example, in PHP, attackers exploit deserialization vulnerabilities using the so-called Property-Oriented Programming (POP) technique, where class properties are reused as *gadgets* and stitched together to construct attack payloads, causing remote-code-execution-like effects on the system [10], [58]. As the use of "gadget" implies, this technique is similar to "gadget"-based code-reuse attacks on native code (e.g., ROP [57]).

```
1  <?php
2  class MyClass {
3    private wrapped_object;
4    /* snip */
5  }
6  class MessageLogger {
7    // __wakeup gets automatically invoked when a
8    // MessageLogger object is unserialized, and
9    // can be used as a gadget when constructing
10   // an exploit
11   public function __wakeup() {
12     unlink($this->logFile);
13   }
14 }
15 // The developer intends to deserialize a
   ↪  MyClass
16 // object, but malicious input can cause a
17 // MessageLogger object to be deserialized
18 // instead, triggering unlink on an arbitrary
19 // path of the attacker's choice
20 $myclass_obj = unserialize($serialized_myclass);
```

Fig. 1: Code showcasing the risks of PHP's native deserialization.

An example of one such vulnerability in PHP is illustrated in the snippet in Fig. 1. The code contains the definitions of two PHP classes: `MyClass` (line 2), and `MessageLogger` (line 6). The `MessageLogger` class also contains the definition of the `__wakeup` method (line 11), which is a special-purpose PHP method that is automatically invoked when an object of the corresponding class is deserialized. Next, the code uses PHP's native deserialization API (i.e., `unserialize` in line 20) to deserialize the value of the `$serialized_myclass` variable, which the developer intended to be a (serialized) instance of the `MyClass` class. However, if an attacker controls the value of the `$serialized_myclass` variable, they can instead provide an input that deserializes to an object of an arbitrary class of their choice. By providing a serialized instance of the `MessageLogger` class and setting the `logFile` property of the serialized object to a path of their choice, the

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.23xxx
www.ndss-symposium.org

attacker can trigger a call to the `__wakeup` method during deserialization, effectively using it as a "gadget", and causing the deletion of an arbitrary file (line 12)[1].

Developer guides and documentation warn against passing untrusted input to deserialization APIs [47], but these warnings are often ignored by developers, resulting in deserialization vulnerabilities being a common issue, especially in web applications. Specifically, the Open Worldwide Application Security Project (OWASP) lists deserialization among the Top 10 vulnerabilities [40], factoring in the danger they pose with the affluence of common vulnerability exposures (CVEs) records citing them. In 2022 alone, 132 CVEs related to deserialization vulnerabilities were published, while the GitHub advisory tracker lists 852 vulnerabilities of class CWE-502 (deserialization of untrusted data) [18]. The latter dataset ranks Java, PHP, and Python as the top 3 affected languages (without normalizing for the number of projects). Unsurprisingly, these vulnerabilities were also weaponized for large-scale attack campaigns by malicious actors, such as APT41 [3].

To protect against deserialization attacks, deserialization APIs allow developers to *restrict* what classes can be deserialized at each API invocation by providing a list of allowed or denied classes[2]. Specifying a list of allowed classes can be cumbersome, since developers must not only specify the intended class of the deserialized object itself, but also the classes of all its transitive fields (e.g., `wrapped_object` in line 3 of Fig. 1), while taking into consideration classes that are defined in different modules to the one containing the invocation to the deserialization API. Similarly, when deserializing a collection of objects (e.g., an array or a set), developers need to specify the possible classes for each element, since failing to specify a single class will cause a runtime error. An alternative approach to protect against deserialization attacks would be to specify a deny list of classes that contain methods that can serve as gadgets. However, the developer would again have to scan the entire application code for such classes, including all dependencies, since missing a gadget class (e.g., `MessageLogger` in Fig. 1) can lead to exploitation. To make matters worse, after the initial (allow or deny) list is composed, any update to the application code or any of the dependencies may require repeating this process in order to keep the list up to date. Perhaps due to this cumbersome process, developers rarely specify allow or deny lists when invoking deserialization functions, which in turn leads to a plethora of vulnerable applications. For instance, among the 5.4M PHP invocations to deserialization APIs on GitHub, only $\sim 0.1\%$ ($4.6K$) specify an allow or deny list.

In this work, we focus our efforts on mitigating deserialization attacks in PHP, a prominent language for developing web applications. We present QUACK, a system that *automatically* mitigates deserialization attacks in PHP applications. QUACK works by inspecting application code, detecting the usage of deserialization APIs, and transforming the code to use the safer allow-list-based variants of these APIs. Our key insight is that by statically examining the code that uses the deserialized object, we can automatically infer a comprehensive list of allowed classes that can be used to significantly reduce the attack surface, relieving developers from the strenuous and error-prone burden of manual code inspection. Moreover, even though we implement QUACK for PHP, we believe that this key idea can be the basis for designing deserialization defenses for other languages, such as Python and Java.

The primary challenge QUACK faces is automatically inferring a precise list of allowed classes. QUACK cannot rely on developer-declared types for the variables receiving the deserialized objects: since PHP is an interpreted language, type information is often not present because declaring types in the code is not supported (PHP version 5) or is optional (PHP version 7 and above). Inferring classes based on how the deserialized objects are used shares similarities with classic type inference, but existing algorithms are unsuitable for QUACK, since they are often designed to infer a precise type for each variable, and usually favor shallow, conservative methods that either infer one precise type, or are unable to infer any types if there are more than one candidate types. As our evaluation shows (Section VIII-D), existing type inference cannot infer the classes for all but one deserialized object.

QUACK solves this inference problem using a novel static analysis process. For each deserialization call, QUACK first constructs a *sound overapproximation* of all classes that are available at the callsite, accounting for the dynamic nature of PHP's class loading logic. Next, it filters the list of available classes by employing a novel interprocedural, context-sensitive, and flow-sensitive algorithm to track how deserialized objects are used, constructing a final list of allowed classes, which is then provided to the deserialization API. The idea is inspired by duck typing in languages such as Smalltalk and Ruby where all that matters, from a type perspective, is that the object responds to the methods called on it, except that QUACK infers the potential types purely statically, before any possible attacks at runtime.

We implemented QUACK and evaluated its effectiveness and performance on a dataset of 15 deserialization vulnerabilities in 10 popular applications including CakePHP, the defacto model-view-controller framework for PHP. Our results demonstrate that QUACK effectively blocks *all* gadgets for 80% of the vulnerabilities. For the remaining vulnerabilities, QUACK greatly reduces the number of gadgets available to an attacker by 84%. QUACK prevents all exploits generated by FUGIO [41], a state-of-the-art (SOTA) POP exploit generation tool, for all evaluated applications in our dataset. Furthermore, we sampled three PHP applications containing calls to deserialization APIs, and submitted anonymized pull requests (PRs) including the suggested fixes generated by QUACK. Both were merged in under 12 hours by their developers, validating QUACK's results. In terms of performance, QUACK carries out its analysis of each vulnerability in 193 seconds on average (max 362 seconds) on a commodity laptop, making it suitable for practical use in DevOps cycles.

II. BACKGROUND

Programmers have needed to serialize abstract data types (ADTs) since the early days of TCP/IP standardization [25]. In the time since, use cases have become more complex, and it has become challenging to balance functionality with ease of use. In the paper describing the design and implementation

[1]Section II-A provides more details on deserialization attacks.

[2]Section II-C provides more details on deserialization defenses

of the Java runtime serialization [63], the authors note that "support for identifying and authenticating classes is a basic requirement and is outside the scope". As the popularity of languages supporting serialization grew, security researchers found ways to turn this small design flaw into a tool for remote code execution.

A. *Serialization and Deserialization*

Programmers often need to save the persistent state of an object and restore it for later use. They may wish to store the object in a database or send an object across a network connection between a client and server to be reconstituted on the remote end; or, for a recently popular example, programmers may share trained machine learning models that are represented as program objects [54], [58]. To facilitate these use cases, managed programming languages typically provide built-in native APIs for object serialization and deserialization.

Broadly, *serialization* refers to the task of converting an object into a representation that can be saved and transmitted, which we refer to as the serialized object. *Deserialization* refers to the complementary task of reading a serialized object and converting it back into an object in a program's executing runtime, which we refer to as the deserialized object. Different programming languages may have different terms for the functions that provide serialization support.

Serialization APIs can differ in the expressive power of the objects that can be serialized and deserialized. Some APIs can only deserialize serialized objects into primitive object types, like arrays, strings, and integers; examples include Python's `json` encoder/decoder module [50] and PHP's `json_encode()` and `json_decode()` functions [44]. Other serialization APIs provide rich and complex object representations that can represent class objects, execute methods, and potentially evaluate arbitrary code, like Python's `pickle` module [51], PHP's `serialize()` and `unserialize()` functions [47], and Java's Object Serialization feature [38]. Even the simpler (de)serialization APIs can be vulnerable to attacks [50], and, unsurprisingly, the more powerful APIs provide attackers with more tools with which to craft more sophisticated exploits.

PHP provides a powerful serialization API that allows programmers to represent class objects and to invoke object methods during deserialization [47]. Fig. 2 shows an example of the output produced by serializing a PHP object, wherein the name of the class, and the values of its properties, are encoded. When the serialized string shown in Fig. 2 (lines 22–24) is passed as a parameter to the `unserialize` function, the function returns an initialized `MessageLogger` object with the properties encoded in the string.

PHP's deserialization API provides programmers with some control over methods that are invoked during deserialization. PHP reserves some method names for *magic* methods, which are special methods that override PHP's default behavior when certain actions are performed on an object [45]. Some relevant magic methods are `__wakeup` and `__sleep`, which are called when an object is deserialized and serialized, respectively, and `__destruct`, which is called automatically when an object no longer has references to it. `__wakeup` can be used for re-initializing the object state in the deserialized object by,

```
1  <?php
2
3  class MessageLogger {
4      private string $logFile = "/tmp/log.txt";
5
6      function __wakeup() {
7          // Reset log file on wakeup.
8          unlink($this->logFile);
9      }
10
11     function logMessage(string $message) {
12         $fd = fopen($this->logFile, "a");
13         fwrite($fd, $message);
14         fclose($fd);
15     }
16 }
17
18 $logger = new MessageLogger();
19 $serialized_object = serialize($logger);
20 print($serialized_object . "\n");
21 // Output:
22 // O:13:"MessageLogger":1:{
23 // s:22:"\x00MessageLogger\x00logFile";
24 // s:12:"/tmp/log.txt";}
```

Fig. 2: Serializing a `MessageLogger` object produces a string that encodes the property values of the object. In this example, the serialization string encodes that the object is a `MessageLogger` object with a `logFile` property set to `/tmp/log.txt`.

for example re-establishing a database or network connection for the new object. By implementing a `__wakeup` method for a class, a programmer ensures that the method is called whenever the object is deserialized.

As an extension to the deserialization API, PHP also provides the ability to embed serialized metadata in PHP Archive (PHAR) files [6] used for distributing PHP applications. In PHP versions prior to version 8.0 , when file system operations (e.g., `file_exists` and `is_dir`) are passed a path to a PHAR file prefixed with the PHAR stream wrapper (e.g., `phar://app.phar`), the serialized metadata is read from the file and deserialized. This can cause unexpected object deserialization when a programmer does not anticipate a file operation being executed on PHAR files.

Deserialization APIs also differ between languages in which classes are available for deserialization, e.g., Java only allows deserializing classes that implement the `java.io.Serializable` interface, while PHP implicitly requires that the class be loaded into the interpreter at the time of the deserialization.

B. *Deserialization Vulnerabilities and Exploitation*

Documentation for deserialization APIs frequently warns programmers to never pass untrusted user inputs into deserialization functions [47], [50], [51]. Despite these warnings, programmers continue to write applications and frameworks that allow user inputs to reach these functions, resulting in hundreds of security weaknesses every year. Fundamentally, when untrusted user inputs are deserialized, arbitrary objects are created and inserted into the program's execution runtime environment. Attackers can leverage the properties of unintended objects to cause the program to reach unsafe states.

The risks posed by arbitrary object deserialization depend on the specifics of how the language deserialization implementation treats deserialized objects, and the power of the deserialization API. Deserialization implementations are broadly susceptible to arbitrary object injection attacks, wherein an unintended object is deserialized and confused for an object of a different type, or with data attributes that should not be permitted. Some deserialization implementations introduce the additional risk of direct arbitrary command execution by providing command execution as a feature of the deserialization routine; for example, the deserialization function may call custom initialization handlers of the deserialized object, or even allow for the execution of code injected in the serialized object.

Exploits for deserialization vulnerabilities fit broadly into to following three categories:

1) **Data-injection attacks**: a deserialized object is of the intended type, but has data attributes set to unintended values. Any deserialization API can be vulnerable to these attacks, even those that only represent primitive types like arrays and strings.
2) **Type-confusion attacks**: a deserialized object is of an unintended type. This category of attacks may also include gadget-based type-confusion attacks, such as Property Oriented Programming (POP) attacks in PHP.
3) **Arbitrary-command-evaluation attacks**: the routine that performs deserialization allows for the evaluation of attacker-provided data, resulting in the execution of arbitrary program commands.

Not every language deserialization API allows for all three categories of deserialization attacks. Generally, data-injection attacks are allowed by any deserialization implementation that does not provide data integrity checks, including many JSON encoding and decoding routines. Type-confusion attacks are allowed by any deserialization implementation that allows for the instantiation of arbitrary objects, like PHP's `unserialize` function. Finally, arbitrary-command-evaluation attacks are allowed only by deserialization implementations that allow code evaluation during deserialization. Python's `pickle` module. For example, PHP's `unserialize` native function allows for data-injection and type-confusion deserialization attacks, while Python's `unpickle` function allows for all three categories of attacks: apart from allowing the first two categories, it also enables arbitrary-command-evaluation attacks by providing functionality to directly evaluate Python code represented by data in the serialized object during deserialization.

1) Data-injection attacks: A data-injection attack occurs when an attacker-controlled string is deserialized into an object of the intended type, but with an attribute data set such that the attacker can cause undesired effects.

For example, recall that the string output in Fig. 2 encodes an object of type `MessageLogger` with property `logFile = "/tmp/log.txt"`. If an attacker can control the string passed to the `unserialize` function, they can change the input string to instead be the one shown in Fig. 3, which encodes a `MessageLogger` object with `logFile = ".htaccess"`. When the string is deserialized, the `__wakeup` method is invoked and the `.htaccess` file—which often contains security-related settings for web applications—is deleted. By being able to control the value of the `logFile` property when a `MessageLogger` object is deserialized, the attacker has an arbitrary-file-delete primitive.

```
O:13:"MessageLogger":1:{
s:22:"\x00MessageLogger\x00logFile";
s:9:".htaccess";}
```

Fig. 3: String that encodes a `MessageLogger` object with attributes `logFile = ".htaccess"`. When this string is deserialized and the `MessageLogger` `__wakeup` method is called, the security-critical `.htaccess` file is deleted.

Generally, opportunities to carry out data-injection attacks are infrequent, because they are dependent on class properties and the application logic that surrounds the vulnerable deserialization routine. In our analysis of exploits against PHP CVEs, we did not observe any data-injection attacks.

2) Type-confusion attacks: Type-confusion attacks are a broad class that generally applies to many kinds of vulnerabilities beyond just deserialization vulnerabilities, like use-after-free vulnerabilities in C and C++ programs [35], [59]. In the context of deserialization vulnerabilities, a type-confusion attack occurs when an attacker deserializes an object that is of an unintended type.

Fig. 1 is an example of a type-confusion attack, wherein a developer intends for the deserialized object to have type `MyClass`, but the attacker instead can create a `MessageLogger` object by controlling the input string to `unserialize`. Unlike data-injection attack, the deserialized object was not intended to be a `MessageLogger` type, and the attack could have been prevented if the intended type of the deserialized object was enforced. To carry out the attack, the attacker only has to ensure that the target class (i.e., `MessageLogger`) is available in the runtime environment at the time of deserialization—i.e., the class is either declared in the module containing the call to `unserialize`, or the module containing the class is included statically (e.g., using `require`), or autoloaded dynamically).

As with data-injection attacks, finding a single class that, when deserialized, provides a powerful enough exploit primitive by itself, is uncommon. Instead, attackers have developed *Property-Oriented Programming* (POP) exploitation technique [12], [14], [41], [53], which works by chaining together functionality from methods of different objects, called *gadgets*. To carry out a POP attack, an attacker first constructs a top-level object of a class containing a *magic* method (e.g., `__wakeup`), which, in turn, may call methods of other nested objects contained as properties in the top-level object. Effectively, this magic method serves as the first gadget that starts execution of the POP chain. Next, the attacker identifies classes implementing methods invoked by the aforementioned magic method, and instantiates objects of these class types as *properties* of the top-level object. When this malicious object is deserialized and the initial magic method is invoked, it will trigger a *chain* of nested type-confusion attacks, each one executing some small component of the attacker's full attack. While POP chains are seemingly complex, security researchers have developed tools to automatically identify candidate POP gadgets to build POP chains [15], [41], [56] and create catalogs of common POP chain gadgets for creating powerful primitives. Essentially, POP exploitation is a specialization of the broader type-confusion category of deserialization attacks.

All PHP deserialization attacks that we observed fall under the type-confusion attack category. Most languages' deserialization implementations are vulnerable to type-confusion attacks, although the specific requirements for a successful attack vary by language. Type-confusion attacks is the class of deserialization attacks that QUACK is designed to mitigate.

3) Arbitrary-command-evaluation attacks: Some languages' implementations of deserialization routines expose powerful APIs to evaluate arbitrary commands. For example, Python's `pickle` module represents serialized objects as bytecode instructions, and implements a virtual machine that executes those instructions when deserializing the object [51], [58]. Notoriously, the virtual machine provides opcodes that interface to call Python's `eval` function on serialized data, which provides a simple yet powerful mechanism for attackers to execute arbitrary code. Most Python deserialization attacks are arbitrary-command-evaluation attacks, because of the attack's simplicity when compared to data-injection or type-confusion attacks.

C. Mitigating Deserialization Attacks

Programming language documentation frequently warns that untrusted user inputs should not be passed directly to deserialization routines. Despite these warnings, deserialization vulnerabilities persist, attracting more and more attention from the security research community. In response, language developers have made attempts to increase the visibility of warnings and to give programmers some control over the types allowed to be deserialized.

As an example, in PHP version 7.0, an optional `allowed_classes` parameter was added to the `unserialize` function to restrict the types of the objects that are allowed to be deserialized at the call-site [47]. This provides an effective mechanism for preventing type-confusion attacks in PHP. However, our analysis shows that this optional parameter is rarely applied, since it requires additional developer awareness and effort. The mitigation is so rarely applied that some POP gadget detection tools do not even bother to model its semantics when searching for potential POP gadgets for attacks [41]. Proper use of the `allowed_classes` parameter applied to the `unserialize` call in Fig. 1 would prevent an attacker from deserializing a `MessageLogger` object:

```
unserialize($data, ['allowed_classes' => ['MyClass']])
```

As a further mitigation, in PHP version 8.0, the default behavior of file operation functions that interact with PHAR file formats (described in Section II-A) was changed so that PHAR metatadata is not automatically deserialized. Instead, PHAR metadata is only deserialized when the programmer explicitly requests it with a `Phar::getMetadata` method, which now also supports an `allowed_classes` optional parameter [49].

Java also provides a mechanism for restricting the allowed classes available at deserialization time, called a *serialization filter* [39]. Unlike PHP, which provides the `unserialize` function and which acts as a single vulnerable call site to protect, Java classes are serialized by implementing the `Serializable` interface. This makes filtering the allowed classes more onerous because each serializable class has its own serialize methods. Java provides two mechanisms for filtering: a) through a JVM-wide filter that is applied to every deserialization in the JVM, and b) through a stream-specific filter that needs to be implemented for the classes they protect. Filters are pattern-based, and so can be either allow-lists or deny-lists. This all results in an interface for filtering allowed classes that is very configurable, but which is very tedious and challenging to implement.

Mitigations like PHP's `allowed_classes` and Java's serialization filters are not frequently applied, resulting in the continued exploitation of deserialization vulnerabilities. One of the reasons for that is the lack of support from integrated development environments (IDEs) and other analysis tools in inferring which classes should be allowed.

D. Inferring Allowed Classes

In programming languages theory [13], the general goal of classic type inference is to check that typing rules are satisfied, or report type errors otherwise. This is typically done by collecting all typing facts such as literals and declarations, and propagating them through data-flow chains to detect conflicts. Classic type inference is thus not targeted enough to infer just the type/class of one deserialized object. In gradually typed languages, like PHP, the effectiveness of classic type inference is limited by the number of facts and type hints in declarations to build upon. For instance, a classic type inference algorithm would collect the `mixed` return type of unserialize as a fact from its declaration, and this fact would never conflict with any other, as `mixed` is a union of all possible types. From our experience, this is exactly what happens in commercial IDEs such as `PhpStorm` [27] and static analysis frameworks such as `noverify` [61]. As we show in our ablation study, only in one case there was a type declaration that allowed for direct type-inference to recommend the relevant allowed classes.

III. THREAT MODEL AND LIMITATIONS

This section describes the threat model according to which QUACK is designed. It also describes the limitations of QUACK when operating in this threat model.

A. Adversarial capabilities

We assume an attacker with full control over a serialized object used as input to the deserialization API. Examples of such attacks include: (1) sending an HTTP GET request to a web application followed by the web application passing some part of the HTTP data directly to a deserialization routine [3], [34], [36], and (2) an attacker uploading a manipulated pre-trained machine learning model which is later deserialized to be used in a system using this model for inference [54], [58].

We also assume the attacker has precise knowledge of the application and library code installed on the attacked entity, allowing them to craft the serialized object to realize an exploit of their choosing.

B. *Hardening Assumptions*

QUACK is designed to stop deserialization-based exploitation performed via type-confusion attacks, leaving data-injection-only and arbitrary-command-evaluation attacks out of scope.

Similarly to the attacker, QUACK requires full and precise access to the application and library code. Specifically, QUACK cannot be used to suggest generic protection for a library without the full context of the application using it.

QUACK does not rely on the PHP runtime to implicitly block exploits from using gadgets in classes that might not be available as they were not loaded in the current session. Instead, QUACK collects the minimal available classes and explicitly bars other classes from being instantiated during the deserialization process.

C. *Limitations*

PHAR As described in Section II-A, in PHP versions prior to version 8.0, the application may be exposed to deserialization vulnerabilities when file operations implicitly deserialize metadata of PHAR file objects. While PHP 8.0+ is not widely deployed yet, applications using PHP 8.0+ are not vulnerable to this attack, since metadata is only deserialized explicitly in the application code. Being future-driven, we decided to leave the implicit PHAR metadata deserialization (for PHP versions before PHP 8.0) out of scope. While it was not part of our evaluation, QUACK can be extended, using the same analysis, to suggest `allowed_classes` for PHAR's `getMetadata` method in the same way that it does for calls to `unserialize`.

Unresolved dynamic behavior QUACK can soundly handle all PHP's static features, and provides partial support for several of its dynamic features while alerting the user when a non-supported case is encountered. PHP has many dynamic features rendering precise static analysis prohibitively complex. Hills et al. [21] provide a useful catalog of PHP's dynamic features, and show that many of these features are not used in practice or are unnecessarily used to implement much simpler static patterns, although some features are used to implement dynamic behavior that can not be resolved statically. QUACK handles magic methods using over-approximation, and provides partial support for dynamic inclusion and registered autoloaders (Section V). In the cases of unsupported imports and dynamic invocations, QUACK issues an alert that stops the analysis and outputs the currently allowed classes as the sound (yet probably imprecise) result. Examples of dynamic invocations are shown and discussed in our evaluation Section VIII-B. Moreover, QUACK performs the allowed classes inference based on the assumption that the use patterns of the deserialized object are available. This assumption is broken for cases where the object is partially updated.[3]

IV. OVERVIEW

Deserialization vulnerabilities are exploited by creating objects with properties that the programmer did not intend to exist in the application. Deserialization APIs create a larger attack surface by allowing more object types to be accessible than is necessary. If the deserialization APIs can be restricted to only allow the programmer intended classes, then the attack surface is significantly reduced. This section provides an overview of how QUACK automatically determines which classes should be allowed during deserialization, in order to restrict the set of classes available to an attacker.

```php
1  <?php
2
3  /* snip */
4
5  $owa = new owa_php();
6  $raw_event = owa_coreAPI::getRequestParam('event');
7
8  $dispatch = owa_coreAPI::getEventDispatch();
9  $event = unserialize(base64_decode($raw_event));
10 $dispatch->notify($event);
```

(a) queue.php

```php
<?php

class owa_eventDispatch {

    function notify($event) {
        owa_coreAPI::debug("Notifying listeners of"
            . $event->getEventType());
        /* snip */
    }
}
```

(b) eventDispatch.php

```php
<?php

class owa_event {

    function getEventType() { /* snip */ }
}
```

(c) event.php

Fig. 4: Open Web Analytics v1.5.6 containing CVE-2014-2294.

A. *Motivating Example*

As a motivating example, we will use a real-world vulnerability (CVE-2014-2294) affecting Open Web Analytics v1.5.6, seen in the snippets in Fig. 4. In this example, a user-provided input (`$raw_event`) is passed to the `unserialize` function (line 9 in Fig. 4a), allowing an attacker to carry out a type-confusion POP attack, as described in Section II-B2. In fact, the FUGIO [41] exploit generation tool is able to synthesize 14 unique exploits against this vulnerable application, all granting the attacker the ability to manipulate files by chaining methods from multiple classes.

If the intended type of the `$event` object could be discerned, then the `allowed_classes` parameter could be used to restrict which classes can be deserialized, potentially removing classes needed by POP chains. However, there is no type information available in the vulnerable code. In order to infer the allowed classes for the deserialized object, additional context needs to be used, by observing how the object is used and what properties are expected of it.

[3]see Fig. 7 for additional information and examples on this topic.

B. *Using* QUACK *to Protect Deserialization APIs*

QUACK is built on the insight that the intended type of the deserialized object can be *inferred* by statically observing how the object is used after deserialization. An overview of QUACK's process is shown in Fig. 5. First, QUACK conducts its analysis step to identify the calls to `unserialize`, and determine the set of available classes at each callsite. Then, in the deserialization protection step, QUACK computes and applies the allowed classes constraints by rewriting each callsite with a safe API call, resulting in the protected application.

In the example in Fig. 4, QUACK is able to determine the set of allowed classes observing how the `$event` object is used. After being assigned the return value of the `unserialize` call, the `$event` object is passed as an argument to the `notify` method of the `$dispatch` object (line 10 in Fig. 4a). This first step of the type inference process can also be performed using a traditional tool, as `owa_coreAPI::getEventDispatch()` is a call to a static method whose return value's type can be inferred, revealing that the `$dispatch` object is an instance of the `owa_eventDispatch` class. The next step requires the use of our novel static duck typing technique. The `notify` method (line 5 in Fig. 4b) accepts `$event` as an argument and calls its method, `getEventType`. However, no other type information is available or can be directly inferred in traditional approaches. On the other hand, QUACK uses the evidence from the call to `$event`'s method `getEventType` to filter the set of available classes and determine that class `owa_event` is the *only* one with method `getEventType`, and therefore that `$event` is intended to be of type `owa_event` (shown in Fig. 4c).

QUACK then rewrites the call to `unserialize` to take the optional `allowed_classes` parameter with the set of allowed classes created during the analysis step:

```
$event = unserialize(base64_decode($raw_event),
['allowed_classes' => ['owa_event']]);.
```

When the `allowed_classes` parameter in this example is set to `['owa_event']`, the 14 POP chain exploits generated by FUGIO are completely mitigated, since they all require classes other than `owa_event` for gadgets in their exploit chains. This effectively mitigates any type-confusion attacks.

As illustrated by this example, QUACK is able to determine the set of available classes at the time of an `unserialize` call, and restrict that set to allowed classes by observing how the deserialized object is used. This allows QUACK to infer the intended type (or set of types) of the deserialized object, and rewrite the unserialize call to also pass the `allowed_classes` parameter with the inferred set of allowed classes, protecting the application from type-confusion deserialization attacks.

V. PROTECTING DESERIALIZATION VIA STATIC DUCK TYPING

In this section, we describe the design of the static duck-typing type inference technique used by QUACK. This technique works by starting from the set of *available* classes at each deserialization callsite, and applying filtering steps based on object use patterns collected from the code to infer the set of *allowed* classes. We first describe a slightly simplified version of collecting all the locations where the deserialized object is used, then detail the PHP-specific rules for collecting evidence from each using location, and finally discuss the required complication to support edge cases.

Algorithm 1: GetAllowedClasses

```
Input: DSCStmt – Deserialization Call Statement
Output: AllowedClasses – The set of allowed classes
1  AllTracked := EmptySet()
2  CurTracked := EmptySet()
3  AllowedClasses := EmptySet()
4  CurTracked += Def(DSCStmt)
5  AvailableClasses :=
     AvailableClassesAtStmt(DSCStmt)
6  while NotEmpty(CurTracked) do
7      Tracked := POP(CurTracked)
8      if Tracked ∈ AllTracked then
9          Continue
10     AllTracked += Tracked
11     foreach CUStmt ∈ Uses(Tracked) do
12         AllowedClasses +=
             EvidenceFromStmt(CUStmt,
             Tracked,AvailableClasses)
13         CurTracked += Def(CUStmt)
```

Deserialization mapping Given an application containing calls to a deserialization API, QUACK first parses all its files, and transforms them into their abstract syntax tree (AST) representation. Traversing the ASTs allows for detecting all call statements that might target deserialization APIs, denoted as DSCStmts. To support cases where the same object is deserialized in multiple locations, and in each location a different part of the deserialized objects is processed, QUACK tracks the uses of all serialized objects. To perform this tracking, QUACK creates a mapping between objects containing a serialized object, to all the points in the program where they are deserialized. After QUACK computes the set of allowed classes for each deserialization call, the mappings will be combined:

$$\text{Deserializations} : \text{SerObjs} \rightarrow \text{DSCStmt} \rightarrow \text{AllowedClasses}$$

To populate Deserializations, QUACK applies the pseudo-code algorithm in Algorithm 1 to all members of DSCStmts. For each member, DSCStmt, Algorithm 1 returns AllowedClasses, the set of classes possibly deserialized at DSCStmt.

Informally, Algorithm 1 performs a comprehensive data-flow analysis, collecting all statements in the program using deserialized data. This collection is achieved by tracking all direct uses, and transitive uses of variables assigned the whole or part of said data, via class field or collection member access. Algorithm 1 uses the Def-Use terminology used in program slicing to express how different statements read from and write into the program's variable, expressed as calls to helper routines (Uses() and Def()) [62]. For example, for $S =$`$a = $b + $c`, $\text{Def}(S) = \$a$, and $\text{Uses}(S) = \$b, \$c$. Other helper routines will be explained next as they are used.

In lines 1–3, three sets are initialized to track: all processed objects (AllTracked), the objects left to process (CurTracked),

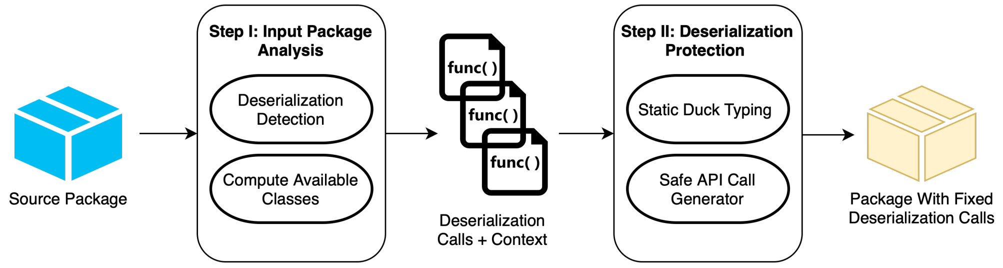
Source Package
Step I: Input Package Analysis
Deserialization Detection
Compute Available Classes
func( )
func( )
func( )
Deserialization Calls + Context
Step II: Deserialization Protection
Static Duck Typing
Safe API Call Generator
Package With Fixed Deserialization Calls

Fig. 5: Overview of QUACK's analysis process. First, QUACK conducts the analysis step to identify calls to deserialization APIs, determine the set of available classes at the callsite, and infer the intended set of allowed classes for the deserialized object. Then, QUACK protects the application by rewriting the vulnerable callsites to restrict the allowed classes and produce safe deserialization calls.

and the resultant allowed classes (AllowedClasses). The first two sets are required to ensure every object is processed only once, thus ensuring termination. In line 4, the deserialization API call statement is analyzed so that the variable representing the newly created deserialized object is added to the CurTracked set to be analyzed first. In line 5, QUACK collects all classes that are available (i.e., can be loaded) at the deserialization callsite (described in Section VI), to populate AvailableClasses. The rest of Algorithm 1 contains a while loop iterating over the depleting and refilled CurTracked set, analyzing them one by one. Lines 7–10 implement the termination assurance described above. In lines 11–13, all the statements using the current tracked object are iterated over (line 11) to extract all usage evidence (line 12) and repopulate as required all other assigned objects (line 13). Usage evidence collection, `EvidenceFromStmt`, accepts 1) the current statement, 2) the object holding the deserialized data, and 3) the set of allowed classes in the deserialization call location that are all used for this process, and returns the set of classes that are possibly legally deserialized and thus must be allowed to deserialize. This output is added to the AllowedClasses set. Next, we provide the details for `EvidenceFromStmt`.

To simplify the explanation, and without loss of generality, in Algorithm 1 we assume the result of the deserialization API is assigned directly to a variable, e.g., `$a = unserialize($input)`. In practice, to support the realistic cases where the returned object is part of a more complex expression, e.g., `$a = (SomeClassCast)unserialize($input)`, we apply a variant of `EvidenceFromStmt` to DSCStmt as well.

Extracting class evidence from statement Algorithm 1 constructs a set of statements that contain evidence of different classes contained in the deserialized object, which will help QUACK construct the set of allowed classes for said object. QUACK performs this class evidence extraction process using a set of pre-defined rules, applying exact or duck-typing matching logic. Table I contains the main rules used by QUACK for this purpose. Every row describes a rule. The first column lists the rule match type (i.e., exact or duck-typing), the second lists the partial statement matching criterion, and the third contains the used classes collected, while t denotes the currently tracked value from Algorithm 1. Exact matching returns a *specific* type for the object and relies on the explicit type being present in the code, such as the (known or deduced) type of a function argument, or an explicit cast of the value to a certain type. For duck-typing matching, QUACK does not return a specific type, but instead returns a set of possible types by filtering the list of available classes (AvailableClasses in Algorithm 1), based on the relevant evidence from the code.

Demands from the underlying static analysis QUACK assumes the existence of an underlying program analysis to perform data-flow tracking and other operations described in Algorithm 1. Specifically, QUACK relies on these tools to detect aliasing and include them in the `Uses` operator. Ideally, the aliasing will be precise, i.e., only including symbols when the relevant value is assigned (possibly by renaming, or a static single assignment (SSA)-based approach). Another option to trade off performance with precision is using a k-bound interprocedural analysis, possibly performed lazily on the parts of the code where a deserialization API is called. When such a k-bound approach is applied, escape analysis can support the case where the k'th procedure is reached to know if the tracking is stopped, and assign the statement a $\top$ value.

Separating evidence by object We now examine how our suggested approach deals with deserialized objects that wrap (i.e., contain references to) other objects. When an object, `$a`, contains a reference to another object, `$b` (stored as an array member or a class property), serializing `$a` (e.g., `$ser = serialize($a)`) will result in `$ser` containing both `$a` and `$b`. Now, consider the following code snippet: `$a = unserialize($ser);$b = $a->somefield; $b->foo()`. This code snippet will trigger two `EvidenceFromStmt` calls: the first call will resolve and collect `$a`'s type (using `somefield`) and start tracking `$b`, and the second will do the same for `$b`'s type (using `foo`). If the classes corresponding to both objects are not specified in `allowed_classes`, the deserialization of `$a` (which contains `$b`) will fail.

Alas, while this approach provides a simple way to deal with wrapped objects, it might introduce false positives. For example, the following code snippet: `$z->zoo();$z->bar()` will result in the first use adding all classes with `zoo`, while the second does the same for `bar`. A more precise result will include only classes that have *both* the `zoo` and `bar` methods. Thus, instead of performing separate evidence collection for each use (as described in Algorithm 1), QUACK collects evidence from all the uses of a specific object together. Namely,

TABLE I: Statement matching rules used by QUACK to collect used classes evidence from statements. Exact or duck-typing rules are applied to the relevant statements to construct the allowed class set.

Rule Type	Partial Statement Matching Rule	Possible Classes
Exact	FunctionX($\text{arg}_1, \text{arg}_{i-1}, t, \text{arg}_{i+1}, ...$)	TypeOf(FunctionX's arg_i)
	ClassXInstance→MethodX($\text{arg}_1, \text{arg}_{i-1}, t, \text{arg}_{i+1}, ...$)	TypeOf(ClassX→ MethodX's arg_i)
	(TypeName) t	TypeName
	Expr ? t : a (or symmetric case)	TypeOf(a)
Duck Typing	t->MethodX(...)	Classes with a method named 'MethodX'
	t.FieldX	Classes with a field named 'FieldX'
	t <BinaryOp>a	Types allowing <BinaryOp>(e.g., "+" or ">=") with TypeOf(a)
	t<Op> (or symmetric case)	Types allowing Op (e.g., "++")
	t[offset or key]	Types compatible with slicing
	a <AssignOp> t	Types allowing <AssignOp>(e.g., +=) with TypeOf(a)
	switch (t): case (a)	Types allowing equality check against TypeOf(a)

for the code snippet above, QUACK will add classes that have `zoo` and `bar` methods to the set of allowed classes.

Dealing with magic methods QUACK deals with PHP's magic methods by over-approximating behavior for classes that implement them. PHP's magic methods allow class authors to add dynamically defined semantics to several basic actions performed on the class. QUACK collects information about classes that contain them, and expands all evidence collection rules to over-approximate them to fit the specific dynamic behavior allowed by the magic method. For example, if a class implements the `__get` magic method, which allows implementing logic for fetching non-existing properties from a class, it will cause any field-matching duck-typing rule (i.e., the second Duck Typing rule in Table I) to return this class as well, even if it does not explicitly contain the fetched property.

Fixing deserialization calls QUACK uses the populated Deserializations map to fix the deserialization APIs in it. Specifically, for each deserialization statement, QUACK restricts the classes that are allowed to be deserialized to the ones in the corresponding AllowedClasses set constructed for the statement.

VI. STATICALLY COMPUTING MINIMAL GUARANTEED DESERIALIZATION-AVAILABLE CLASSES FOR PHP

In this section, we describe the class loading process performed by the PHP interpreter, and how we use its rules to calculate the minimal set of classes that are guaranteed to be available at different locations of a PHP program. This calculation provides the initial class set used by QUACK's static duck typing inference technique described in Section V.

A. Class loading in PHP

For a given file, besides the classes defined in it, PHP provides two methods for loading[4] additional classes defined in other files.

The first method is explicit, using the `include` or `require`[5] directives [43]. These directives, followed by the path to the file to include, cause the mentioned file to be parsed, and all classes in it to be loaded. Note that include directives and classes can be specified anywhere in the file, and loaded either as a part of the initial include directive, or as a part of a follow-up dynamic execution of part of the code.

[4]We use the term loaded for a class to mean the class definition was parsed and its code was executed by the interpreter

[5]For brevity we will henceforth use include to mean include or require

Secondly, PHP also allows developers to specify custom dynamic autoloaders [42], which might trigger an implicit class loading later. When a class that is not currently defined (i.e., has not yet been loaded) is referenced, PHP will call all currently registered autoloaders, which will try to dynamically locate and load the file containing the class definition. Even though autoloaders can implement arbitrary logic for locating a file containing the definition of a non-defined class, autoloaders are often implemented according to specifications published by the PHP Framework Interop Group [2], the most common one being the PSR-4 [48] specification.

An autoloader implementing the PSR-4 specification attempts to autoload classes based on a mapping between a PHP namespace [46] (e.g., `MyNamespace\Foo`) and a project subdirectory (e.g., `src/dir`). When a non-defined class belonging to the specified namespace is referenced (e.g., `MyNamespace\Foo\MyClass`), the autoloader will look for a PHP file matching the class name under the specified subdirectory (e.g., `src/dir/MyClass.php`). If such a file exists, the autoloader will load it using the `include` directive. If the class contains sub-namespaces (e.g., `MyNamespace\Foo\Bar\MyClass`), the sub-namespaces are converted to subdirectories (e.g., `src/dir/Bar/MyClass.php`). A PSR-4 autoloader is provided by the Composer dependency manager [1], which is a commonly used dependency management tool for PHP projects. A project using Composer defines mappings between namespaces and directories in a JSON file, and Composer automatically generates an autoloader in a file named `vendor/autoload.php`. The generated autoloader loads classes based on the PSR-4 specification using the developer-specified mappings.

B. The class-def-graph

QUACK creates the class-def-graph (CDG) to *statically* compute an over-approximation of the classes that can be loaded by the interpreter at every line of the application. This set includes not only all the classes that were *intended* to be deserialized, but also all the classes that are *available* to an attacker at each deserialization callsite, and can be used to carry out the type-confusion attack described in Section II-B2. Thus, the soundness of the analysis creating this set is crucial to the success of QUACK's operation. The CDG is a directed graph, whose nodes represent files in the code, annotated with classes

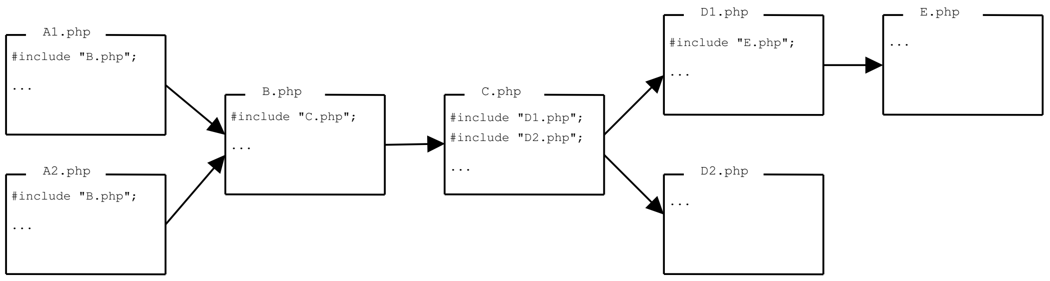

Fig. 6: QUACK constructs and traverses the class-def-graph (CDG) to statically compute the available classes at each program location.

loaded by these files, and its edges represent dependencies on other files using either explicit or implicit connections. Dealing with class definitions and `include` directives, while not very common in practice, is handled by QUACK by considering all classes and recursively following all include directives regardless of their location in the file. Fig. 6 presents a CDG graph, which we will use as a running example to explain QUACK's process for creating an CDG, and using the CDG to determine the set of available classes at each deserialization callsite. The graph contains seven files: `A1, A2, B, C, D1, D2`, and `E` (all with a `.php` postfix).

Constructing the CDG To construct the CDG, QUACK first adds nodes to the graph, and then analyzes the code to add the edges. First, QUACK parses each file into its AST representation, and looks for AST nodes representing `include` or `require` directives. For each directive, the argument containing information about the path to include needs to be resolved into a file path that is the target of the directive. If the argument is a static string literal (or composition of these), it is used as is. Otherwise, QUACK attempts to resolve the path using pattern matching and rule-based approaches suggested by previous work (e.g., Saphire [8]). For example, if the path contains any references to constants or variables, QUACK replaces them with the AST node representing the referenced constant/variable. Furthermore, if the included path contains function calls, QUACK attempts to reproduce the result of the call over the argument in the case where the call is to a known API (e.g., built-in PHP string manipulation APIs). QUACK keeps replacing the nodes recursively until the include path only contains either string literals, or nodes that cannot be resolved (e.g., a call to an unknown API). Finally, QUACK creates a regular expression by concatenating the resulting string literals and replacing unknown nodes with a wildcard (*). Finally, the regular expression is evaluated, and any matching project files are recorded as the resolved dependencies for the file.

When encountering autoloaders, QUACK examines their calling context. If the autoloader is a Composer-generated PSR-4 autoloader, QUACK parses the developer-specified namespace-to-subdirectory mappings, and uses its own PSR-4-compliant implementation to detect all possible files that can be autoloaded based on the specified mappings. QUACK then adds an edge between the file including the autoloader (i.e., `vendor/autoload.php`) and each detected file. Other autoloader schemes are not supported and lead to an error.

In the example in Fig. 6, QUACK will construct the CDG as presented in the figure: it will create edges from `A1.php` and `A2.php` to `B.php`, from `B.php` to `C.php`, from `C.php` to `D1.php` and `D2.php`, and from `D1.php` to `E.php`. Note that the CDG contains *dependency paths* between files not only when a file explicitly includes another, but also when there is an *implicit* dependency (e.g., there is a path from `A1.php` to `E.php`).

Traversing the CDG When a call to a deserialization API is made from within a file, all classes defined in all files located on a dependency path that includes the file containing the deserialization will be available to the PHP runtime. Given the file path containing the deserialization call, F, QUACK traverses the graph and constructs this set of available classes. QUACK starts traversing at the node representing F, and makes a forward and then a backward pass. For the forward pass, QUACK recursively follows all outgoing edges and records all classes defined in all traversed files. Similarly, for the backward pass, it follows all the incoming edges starting from F, and collects all classes defined in the traversed files as well. Using the CDG in Fig. 6, we can follow the process of traversing the graph for a deserialize call located in `C.php`. The forward pass will traverse `D1.php`, `D2.php`, and `E.php`, and collect all classes defined in these files. The backward pass will traverse `B.php`, `A1.php`, and `A2.php`, and collect all their classes, constructing the final available class set.

VII. IMPLEMENTATION

We implemented QUACK as an extension of the analyses provided by the Psalm and Joern program analysis frameworks. Psalm is a PHP static analysis tool primarily used for providing linting information for PHP source code; our first implementation forked Psalm version 5.7.0 [60] and added $\sim$ 3000 lines of code to implement QUACK's static duck typing algorithm. While Psalm supports all versions of PHP, we did encounter cases where Psalm's (unmodified) parsing or analysis passes led to crashes or out-of-memory errors. To overcome some of the analysis limitations of Psalm, we complemented Psalm's analysis by extending QUACK to use Joern. Joern is a static analysis framework that supports the analysis of multiple source code programming languages, to include C, Java, and PHP, and enables security researchers to query source code for vulnerability patterns [65]. We forked Joern version 2.0.140 and extended it to support type inference analyses for PHP source code, and implemented QUACK's static duck typing algorithm as a Scala program that interacts with Joern's analysis API. To facilitate the analysis of WordPress plugins, we employed the `wordpress-stubs` PHP package [32]. For our CDG-based available classes computation process, we used parts of Saphire [8]. Unfortunately, the Saphire framework only supports analysis of PHP code version < 8.0, which further limited QUACK's analyses.

VIII. EVALUATION

We developed QUACK to mitigate the exploitation of deserialization vulnerabilities in managed programming languages. We intend for QUACK to be practical for developers to apply to production applications and frameworks at scale. To determine the efficacy of QUACK, we evaluated it along three research questions:

RQ1: How well does QUACK reduce the exploitability of deserialization vulnerabilities?

RQ2: What is the analysis runtime of QUACK?

RQ3: How do the different components of QUACK contribute to its success?

A. Experiment Setup

Our evaluation is composed of three PHP datasets: **FUGIO**, **VULN202X**, and **CRAWLED**.

The **FUGIO** dataset was adapted from the dataset used by the authors of FUGIO [41] to evaluate the effectiveness of their system in generating exploit chains against deserialization vulnerabilities. The dataset used to evaluate FUGIO is composed of PHP applications, each with an identified deserialization vulnerability, and includes all applications used by Dahse et al. [12] in their evaluation. Some vulnerabilities in the dataset are from real disclosed CVEs, while others are artificially introduced in order to test the efficacy of FUGIO at generating exploits. Among the real CVEs in the original dataset, some vulnerabilities were caused by user inputs reaching the `unserialize` function, while others were PHAR vulnerabilities resulting from PHP file operations on serialized metadata. After removing the artificial vulnerabilities (for lacking enough context for meaningful analysis), the PHAR vulnerabilities (for lacking the `allowed_classes` mechanism for mitigation), one misclassified vulnerability (CVE-2014-0334 is an XSS in `CMS Made Simple`), and vulnerabilities for which FUGIO was not able to generate exploits in their original evaluation, we were left with five vulnerable applications to evaluate.

Aiming to experiment on a larger set of vulnerable applications, specifically ones targeting modern PHP versions, we explored the vulnerabilities in the NIST national vulnerability dataset [37] looking for deserialization vulnerabilities to construct **VULN202X**. Specifically, we sampled 12 CVEs from all CVEs awarded between the beginning of 2020 and the first half of 2023. Following a manual examination we had to exclude 7 that were PHAR-based, wrongly classified (e.g., CVE-2022-48093 which is a command injection vulnerability), did not contain enough information to identify the vulnerable code location (e.g., CVE-2022-33900), or involved software that is not open sourced (e.g., CVE-2023-25135 for vBulletin), leaving five CVEs in the **VULN202X** dataset.

Finally, we crawled Github looking for PHP projects using deserialization APIs to create **CRAWLED**. Applying QUACK to random projects allowed us to put QUACK to the test of analyzing real-world projects in an uncontrolled scenario and also allowed us to contribute to the open-source community. Specifically, we used the query in [17] also including a filter for "stars > 100" to download five random projects.

Evaluation environment We run our experiment on a Linux workstation with an eight-core Intel i7 CPUs @ 1.90GHz and 16GB of RAM.

B. RQ1: How well does QUACK reduce the exploitability of deserialization vulnerabilities?

We designed QUACK to mitigate the exploitation of deserialization vulnerabilities, leaving the existing semantics of the application unchanged. In light of this goal, we perform two experiments to evaluate QUACK along three complementing axes: (i) gadget blocking, (ii) wrongfully excluded classes and (iii) preventing exploitation. For (i) we measure the proportion of gadgets blocked by QUACK's fix compared to the number of gadgets that exist in the application. For (ii), we compare the set of allowed classes constructed by QUACK against the version adapted by the application's developers. For (iii), we measure if the application protected by QUACK is exploitable in practice, by trying to construct a viable exploit for it.

Measuring all axes is important, as the fact that automatic tools can not construct an exploit does not mean the application is in fact not exploitable, nor that the suggested allowed classes are not missing a required class.

Gadget blocking In Section II-B, we explained that in order to create an exploit, an attacker needs to chain one or more gadgets. As the success of an exploit creation process depends on the payload in mind and the chances of the right primitives being in the right place, we will use the number of gadgets as a proxy measure for how likely is it for the attacker to successfully create an exploit, keeping in mind that no exploits can be created when no classes are allowed.

Table II shows the results of our experiment designed to measure QUACK's fix precision. Each row in Table II contains information about one vulnerability, detailing the source dataset, CVE, application name and version, and the fix precision details. As CVE-2021-25294, reported against OpenCATS, contained five independent deserialization API vulnerabilities, we present each vulnerability in its own row. The number of gadgets that were available for exploit creation before applying the fix suggested by QUACK is listed as blocked gadgets, while the ones that remain after the fix are listed as remaining gadgets. We also specify the percentage of gadgets blocked proportional to the total gadget count in the application. We decided to measure gadgets compared to the upper bound, all gadgets in the application, as the specific classes loaded and kept in the interpreter's memory depend on previous executions and implementation details of the interpreter.[6]

On average, using QUACK's suggested fix succesfully blocks 97% of gadgets. For 80% of cases, all gadgets are

[6] [31] shows an interesting vector for tricking PHP into loading classes that can later be used as part of a POP exploit chain.

TABLE II: Applying QUACK's suggested fix to applications with deserialization vulnerabilities and confirmed fixes, allows, on average, for blocking 97% of gadgets. In 80% of the cases, 100% of gadgets are blocked.

Dataset	Application	CVE	Gadgets # Blocked	# Remaining	% Blocked
FUGIO	Piwik 0.4.5	CVE-2009-4137	115	0	100%
	Joomla-3.0.2	CVE-2013-1453	74	0	100%
	CubeCart 5.2.0	CVE-2013-1465	42	0	100%
	Open Web Analytics 1.5.6	CVE-2014-2294	14	0	100%
	Contao CMS 3.2.4	CVE-2014-1860	150	0	100%
VULN-202X	ForkCMS 5.8.3	CVE-2020-24036	221	23	91%
	WP-hotel-booking 10.2.1	CVE-2020-29047	103	0	100%
	OpenCATS-0.9.5 (1)	CVE-2021-25294	288	0	100%
	OpenCATS-0.9.5 (2)		232	56	81%
	OpenCATS-0.9.5 (3)		288	0	100%
	OpenCATS-0.9.5 (4)		288	0	100%
	OpenCATS-0.9.5 (5)		232	56	81%
	WP-AIOSEO 4.1.0.1	CVE-2021-24307	23	0	100%
	WP-booking-calander 9.1.1	CVE-2022-1463	96	0	100%
	WP-lead-generated 1.23	CVE-2023-28667	40	0	100%

blocked. For the three remaining cases, two from OpenCATS (CVE-2021-25294 (2) and (5)) and ForkCMS (CVE-2020-24036) QUACK blocks an average of 84% gadgets.

The three cases listed above, where QUACK was not able to block all gadgets, have the same root cause: QUACK's static analysis encountered a dynamic invocation, i.e., a call to a procedure called by name, where the name is resolved dynamically. In all these cases, QUACK's analysis raises an alert when such unsupported calls are encountered, as without resolving the dynamically determined call, the analysis can not proceed with collecting more usage evidence. Nonetheless, for this experiment, we decided to force QUACK to continue the execution, causing it to fall back to an unfiltered set of available classes.

Specifically, the encountered calls are PHP dynamic invocation built-ins designed to enable a call-back pattern. For ForkCMS, this was a call to `call_user_func_array([$class, $method],...)` using a class and method names determined by strings fetched from a configuration file designed to allow a plugin-like extension. For OpenCATS, in both cases, the control flow arrives at a call to `new $class(...)`, where `$class` contains a string with the class name. Even though for these cases QUACK could not conclude the analysis, the suggested fix for these three cases still managed to block 84% of gadgets.

Wrongfully excluded classes Over-limiting the allowed classes might protect from attacks, but this could lead to the application crashing when it tried to deserialize a benign class. Thus, we compare the set of classes suggested by QUACK to the ground truth set expressed in the application's fix. When the application's developers fixed the deserialization API call by specifying the `allowed_classes` argument we used that set as is. In other cases, where the developers converted the deserialize calls into a non-native scheme that does not support complex objects (e.g., JSON based) we treated it as if the allowed classes set should be empty. Examining the suggested classes in QUACK's fix, none of the cases caused a legitimate benign class to be wrongfully blocked.

Preventing exploitation QUACK prevents exploitation by restricting the set of class objects that can be used to construct exploits in deserialization attacks. To evaluate how well QUACK prevents spurious class objects from being used in deserialization attacks, for all the applications in our **FUGIO** dataset, we wanted to generate exploits attacking both (1) the original "un-protected" version of the application and (2) the protected version created using QUACK. We then attempted to use all exploits against (1) and (2).

TABLE III: The number of exploits generated by FUGIO [41] against the unprotected and QUACK-protected applications. After applying QUACK's fix, all applications are no longer exploitable.

Application	CVE	# FUGIO-Generated Exploits Unprotected	Protected
Piwik 0.4.5	CVE-2009-4137	1	0
Joomla-3.0.2	CVE-2013-1453	2	0
CubeCart 5.2.0	CVE-2013-1465	1	0
Contao CMS 3.2.4	CVE-2014-1860	5	0
Open Web Analytics 1.5.6	CVE-2014-2294	14	0

We employ the exploit generation tool FUGIO to generate all the exploits for this experiment. FUGIO [41] is a state-of-the-art automatic exploit generation tool for crafting PHP object injection exploits. Given a deserialization vulnerability, FUGIO analyzes the application to determine possible exploit chains. When QUACK's analysis is applied to a vulnerable application, tools like FUGIO should be unable to generate viable exploit chains. We set a timeout of 12 hours for every FUGIO run. We limited this experiment to the **FUGIO** dataset, as the FUGIO requires a triggering input to drive exploiting attempts, which was not available in the right form for the other applications.

We modified FUGIO to properly account for the optional `allowed_classes` parameter to the PHP `unserialize` function. In its original implementation, FUGIO did not account for the semantics of the optional parameters to `unserialize`, which caused it to create exploit chains that may not have been viable in cases where the application code restricted the classes of the unserialized object. Our modification adds the semantics of the `allowed_classes` parameter, in order to make FUGIO's analysis more precise (i.e., make it only consider gadgets in classes contained in `allowed_classes`), and remove false positives. We will make the improved version of FUGIO available.

Table III shows the results for all the other applications. Each row represents two runs of FUGIO, one (left) on the original unprotected application, and another (right) on the QUACK protected one. For each application, FUGIO was able to produce at least one exploit for the unprotected version, and overall results are consistent with the ones presented in the FUGIO paper for these applications.

On the other hand, for the rest of the applications, after applying QUACK's fix, FUGIO was not able to produce any exploits. This is due to the fact that in these cases a) `allowed_classes` was set to `false`, and b) the allowed classes do not contain gadgets (e.g., `owa_event` mentioned in Section IV). Unsurprisingly, trying to execute the exploits generated for the unprotected version against the protected version of the application did not succeed. As an illustrative example, FUGIO identified 14 exploit chains for use against CVE-2014-2294 in Open Web Analytics. However, QUACK determined that at the vulnerable `unserialize` call site, the only class that should be allowed for deserialization is class `owa_event` (also discussed in Fig. 5), which means that any deserialization exploits should only able to use objects of type `owa_event`. Of the 14 exploits created by FUGIO, none contained the `owa_event` class object, and all used other object types. Therefore, all 14 exploits created against the unprotected version would be prevented by setting the `allowed_classes` parameter to `owa_event`.

Submitting fixes to open-source projects Tasking QUACK with analyzing and fixing applications from CRAWLED allowed us to submit three PRs that were merged by their developers. Unlike the experiments on the other datasets, this experiment formed an uncontrolled study, as we did not know if the fix generated by QUACK was correct. Out of the five PHP applications in this dataset, we had to discard two projects that use PHP 8 features (which one of our underlying components lacks support for), yet for the other three, which included CakePHP, QUACK terminated successfully and their results were manually validated to the best of our abilities (examined by two PhD students or more). Turning our attention back to the projects themselves, we were happy to learn they are active (committed to in the last year) and had relevant policies in place to accept pull requests and security notifications from external developers. We submitted QUACK's results along with relevant explanations for the developers about the risks involved and the process we performed to generate the fix as PRs. All PRs were eventually merged by their developers. One example of such a PR can be found here: https://github.com/cakephp/cakephp/pull/17162[7].

C. RQ2: What is the analysis runtime of QUACK?

For QUACK to be practical for developers to use, it needs to be easy to incorporate into existing workflows. The time that QUACK takes to analyze an active project must be small enough that developers can apply it to large projects. We measured the analysis runtime of QUACK on all applications we evaluated and also recorded the number of PHP source code files and approximate lines of code in the application.

Our results, shown in Table IV, demonstrate that QUACK is able to quickly analyze large-scale applications without imposing a burden on developers. All analyses complete within approximately six minutes (362 seconds) and in an average of 193 seconds, which makes QUACK practical to incorporate into developer workflows, whether alongside programming (e.g., linters) or at the end of a development cycle and prior to deployment (e.g., in a Continuous Integration/Continuous Deployment pipeline). All analyses were run in single-threaded mode, due to limitations of the analysis framework that we built QUACK on, using a commodity laptop with 16GB of RAM. This further demonstrates the practicality of QUACK for regular development tasks.

[7]This PR is fully anonymized to allow sharing it as part of this draft.

TABLE IV: Analysis time of QUACK on each analyzed application, with the number of PHP files and approximate lines of code in each application.

Application	Analysis Runtime (s)	# of Files	KLoC
Piwik 0.4.5	38	787	176
Joomla-3.0.2	129	1574	284
CubeCart 5.2.0	52	864	130
Open Web Analytics 1.5.6	39	486	81
Contao CMS 3.2.4	172	583	203
ForkCMS 5.8.3	39	837	101
WP-hotel-booking 10.2.1	276	169	23
OpenCATS 0.9.5-3	270	355	137
WP-AIOSEO 4.1.0.1	362	370	64
WP-booking-calendar 9.1.1	207	88	54
WP-lead-generated 1.23	222	23	5

D. RQ3: How do the different components of QUACK contribute to its success?

In this experiment, we decompose QUACK, creating several variants, aimed at gauging the importance of each component in an ablation study. To measure the importance of each component, we create several variants of QUACK by disabling key components (also creating the relevant pass-through logic) and using the rest of the system as is. To help us describe QUACK's variants we will use the component names depicted in Fig. 5.

We constructed the following variants of QUACK:

- **QUACK-ONLYTYPE**: In this variation of QUACK we disabled both the "Static Duck Typing" and "Compute Available Classes" components, leading to QUACK using the types deduced by the underlying type inference tool if these existed, and otherwise allowing all classes in the application.
- **QUACK-NODUCKS** In this variation of QUACK we disabled the "Static Duck Typing" component, leading to QUACK using the types deduced by the underlying type inference tool if these existed, and otherwise falling back to the collected available classes.
- **QUACK-NOAVAIL**: In this variation of QUACK we disabled the "Compute Available Classes" component, causing all classes in the application to be considered by the analysis.

Table V shows the results of this experiment. Each row presents the blocked and remaining gadget count for one application for all of QUACK's variants. As the applications are the same, due to space constraints, we only keep the

TABLE V: The results of our ablation study (RQ3) measuring the blocked and remaining gadgets for three variants of QUACK compared with QUACK. In each variant one or two key components is disabled.

Dataset	Application	QUACK-ONLYTYPE			QUACK-NODUCKS			QUACK-NOAVAIL			QUACK		
		# Blocked	# Remaining	% Blocked	# Blocked	# Remaining	% Blocked	# Blocked	# Remaining	% Blocked	# Blocked	# Remaining	% Blocked
FUGIO	Piwik 0.4.5	0	115	0%	111	4	96.52%	115	0	100%	115	0	100%
	Joomla-3.0.2	0	74	0%	59	15	80%	74	0	100%	74	0	100%
	CubeCart 5.2.0	0	42	0%	1	41	2.38 %	42	0	100%	42	0	100%
	Open Web Analytics 1.5.6	0	14	0%	2	12	14.29%	14	0	100%	14	0	100%
	Contao CMS 3.2.4	0	150	0%	0	150	0%	150	0	100%	150	0	100%
VULN-202X	ForkCMS 5.8.3	0	244	0%	221	23	90.57%	0	244	0%	221	23	90.57%
	WP-hotel-booking 10.2.1	0	103	0%	103	0	100%	103	0	100%	103	0	100%
	OpenCATS 0.9.5-3 (1)	0	288	0%	56	232	19.44%	288	0	100%	288	0	100%
	OpenCATS 0.9.5-3 (2)	0	288	0%	232	56	80.56%	0	288	0%	232	56	80.56%
	OpenCATS 0.9.5-3 (3)	0	288	0%	56	232	19.44%	288	0	100%	288	0	100%
	OpenCATS 0.9.5-3 (4)	288	0	100%	288	0	100%	288	0	100%	288	0	100%
	OpenCATS 0.9.5-3 (5)	0	288	0%	232	56	80.56%	0	288	0%	232	56	80.56%
	WP-AIOSEO 4.1.0.1	0	23	0%	0	23	0%	23	0	100%	23	0	100%
	WP-booking-calander 9.1.1	0	96	0%	94	2	97.92%	96	0	100%	96	0	100%
	WP-lead-generated 1.23	0	40	0%	40	0	100%	40	0	100%	40	0	100%

application name in this table. As we reported in the previous experiment, QUACK's analysis is conservative in that it does not miss required classes. In this experiment, we encountered the same results for QUACK's variations, i.e., none of the variants cause missed required classes.

Out of all the vulnerable applications, QUACK-ONLYTYPE can only correctly infer the set of allowed classes for one vulnerable location in the OpenCATS application (CVE-2021-25294 (2)), and for the rest of the applications it can not block any gadgets, amounting to an average of gadget blocking percentage of 6.67%. This result unequivocally shows that relying on partial types inferred from the code can not assist with protecting applications against deserialization attacks.

Contrasting QUACK-NODUCKS's results with QUACK shows that it only matches QUACK's success for the three cases we discussed in the RQ1 experiment, where the analysis encounters an explicit dynamic call into a class method using unknown classes and methods. This tracks with the fact that QUACK falls back to allowing available classes in these cases. QUACK-NODUCKS achieves an average gadget blocking percentage of 58.8%. These results showcase the importance of our novel static duck typing approach.

The complementary result to QUACK-NODUCKS is shown in QUACK-NOAVAIL: in the mentioned three cases, QUACK falls back on the full set of classes in the application, dropping its average gadget blocking percentage to 80.1%, compared with 97% achieved by QUACK. This result shows the importance of the static calculation of the available classes as a good fallback option.

IX. DISCUSSION

A. Updating Serialized Objects

Apart from the group of samples we discussed in Section VIII-B, where a dynamic call to a function disrupts QUACK's ability to track the usage of the unserialized object, there is one more group of samples QUACK has limited ability to handle: object updates. In many applications, objects are serialized with the purpose of storing them in a database. When the objects require an update, they are fetched from the database and are deserialized before the update is applied to them, followed by reserialization of the object for storing back to the database. Attackers with even limited access to the database can use such an operation to pivot from the database to the application server by placing an exploit in place of the benign serialized object. Because these deserialization calls are usually not exposed directly to users they are not awarded CVEs, but still put applications at risk.

```php
1  <?php
2  /* snip */
3  private function onEmailSettings(){
4    $mailerSettings = new MailerSettings($siteID);
5    $mailerSettingsRS = $mailerSettings->getAll();
6    $m = unserialize($mailerSettingsRS['message']);
7    $m[STAT] = (UI::Checked('stat', $input) ? 1 :
   ↪   0);
8    /* similar field updating lines follow */
9    $mailerSettings->set('message', serialize($m));
10   /* snip */
11 }
```

Fig. 7: A code example updating a serialized object stored in a database. For this and similar examples, QUACK lacks a global context of all objects serialized in the program to correctly identify all types in the object.

Fig. 7 shows an example of this pattern, which is a simplified version of the code used by OpenCATS version 0.9.7.2. In lines 4–5, the mailer settings are read from the database where they were previously stored in a serialized form, and in line 6 they are deserialized. In line 7 (and the following lines we omitted for space) the settings are updated according to the new input. In line 9 the object is then reserialized, and in the following lines (omitted for space) the serialized object is written back to the database.

When QUACK analyzes this code it faces two problems. First, the origin of the object can not be tracked, as the SQL query used to fetch the serialized object is not the same one used to place it in the database originally. On the other hand, once the deserialized object is used as an argument to `serialize()`, it does not allow the system to learn anything else about possible classes stored in the other fields in the array, and all statements at lines 10 and onward manipulate the serialized object, and not the object itself. We plan on addressing these kinds of use patterns for serialized objects in future work.

B. Expanding QUACK To Additional Languages

We explored vulnerable applications in other languages with the aim of expanding QUACK to support them. As mentioned in Section I, the most prominent other languages suffering from deserialization vulnerabilities are Java and Python. Expanding QUACK to support Java and other statically typed languages will require resolving over-broad type declarations, (e.g., Java's `Object`) into more for specific types for concrete applications. On the surface, expanding QUACK to Python seems more straightforward, as they are both dynamically interpreted languages. Alas, as we mentioned in Section II-B, Python's unique serialization approach that is implemented as a stack-based virtual machine complicates the process of hardening the deserialization process. After we implemented a generic `allowed_classes`-like mechanism, we found out that the expressiveness of the virtual machine language allowed for additional ways to load and manipulate objects, not possible in the static binary-based deserialization methods. We plan on continuing research in this domain.

X. Related Work

Automated deserialization exploit generation Automatically generating payloads for exploiting deserialization vulnerabilities has been explored by several works. Cao et al. suggested ODDFUZZ [9] for using structured information for generating POP exploits for Java projects with deserialization vulnerabilities. `ysoserial` [15] is a tool that generates exploit objects using *known* POP gadgets in specific Java applications. A recent survey [55] studied the exploitability of deserialization vulnerabilities in 19 Java projects using `ysoserial`, and found all projects had enough available gadgets to construct exploits. Dahse et al. [12] examined the exploitability of PHP deserialization vulnerabilities by introducing an automated approach for statically detecting and generating exploitable POP gadget chains. FUGIO [41] improves automated exploit generation for deserialization vulnerabilities by leveraging both static and dynamic analyses, as well as fuzzing, to generate exploit objects that automatically trigger discovered POP gadget chains. We leveraged exploits generated by FUGIO to evaluate the effectiveness of QUACK at preventing deserialization attacks. PHPGGC [56] applies a similar approach to `ysoserial` but targets PHP. The aforementioned works emphasize the importance of QUACK; not only do deserialization vulnerabilities have serious security implications, but exploit generation can also be automated, further increasing the chances of exploitation.

Hardening PHP applications Prior work has attempted to protect PHP applications using *debloating*. Minimalist [26] debloats PHP applications by leveraging access-log files to determine files accessed by users during interaction with the application and using static analysis to perform a reachability study to determine what non-reachable code can be removed from the application. Less is More [7] leverages profiling techniques to collect coverage information for a PHP application and uses the collected information to debloat the application. Saphire [8] limits the set of system calls a PHP application can use by composing a set of the system calls required by the application with a combination of static and dynamic analysis. Unlike the above works that apply a global effort at *reducing* the attack surface against PHP applications, QUACK focuses its effort on the intrusion point and directly attempts to hinder exploitation by only allowing the required classes.

Type inference for dynamic languages Several works have explored type inference via means of (either static or dynamic) program analysis for dynamically typed languages such as Ruby [4], [16], [28], [29], JavaScript [5], [19] and Python [20], [24], [33], [64]. The HipHop compiler [66], a tool for translating PHP code to C++, leveraged a constrained-based algorithm to statically infer types for PHP. It was deprecated in favor of HHVM [23], a virtual machine for the Hack [22] programming language, a PHP variant that supports static typing. Even though HHVM supports types inference, it only supports programs written in Hack, and not regular PHP, and would thus not be suited for use by QUACK. As we discussed above and saw in the evaluation, these tools are too conservative and aim at inferring the exact type, making them unsuitable in most cases for inferring the classes that should be allowed in the deserialization point.

Other works have explored type inference of dynamically typed languages using AI-assisted tools. JSNice [52] uses machine learning to generate type annotations for JavaScript code. Similarly, PYInfer [11] leverages a deep-learning model to statically infer types in Python programs. Finally, Klingström and Olsson [30] explore the use of deep learning to generate type annotations for PHP. Even though AI-based approaches can be useful in applications where soundness is not crucial, that is not the case for QUACK, where unsound type inference could potentially cause crashes by disallowing correct classes from being unserialized.

XI. Conclusion

In this paper, we presented QUACK, a framework for automatically mitigating deserialization attacks. QUACK statically derives the classes that are available at each deserialization API callsite, and leverages static duck typing to filter the classes down to a set of classes that should be allowed to be deserialized. QUACK uses this set to fix deserialization calls in order to prevent all non-allowed classes from being deserialized, effectively blocking gadgets contained in these classes from being used to construct exploits. We implemented QUACK for PHP, and demonstrated its effectiveness at protecting applications against exploitation attempts. QUACK reduces the average amount of exploits generated by a state-of-the-art automatic exploit generation tool from five to zero, by blocking an average of 97% of classes that could be used as gadgets.

Acknowledgment

Acks.

References

[1] "Composer: A Dependency Manager for PHP," https://getcomposer.org/.

[2] "PHP Framework Interop Group," https://www.php-fig.org/.

[3] M. Alyssa Rahman, "Now You Serial, Now You Don't — Systematically Hunting for Deserialization Exploits," https://www.mandiant.com/resources/blog/hunting-deserialization-exploits.

[4] J.-h. D. An, A. Chaudhuri, J. S. Foster, and M. Hicks, "Dynamic Inference of Static Types for Ruby," in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2011, pp. 459–472.

[5] C. Anderson, P. Giannini, and S. Drossopoulou, "Towards Yype Inference for JavaScript," in *European Conference on Object-Oriented Programming (ECOOP)*, 2005, pp. 428–452.

[6] T. Andre, “PHP RFC: Don’t automatically unserialize Phar metadata outside getMetadata(),” https://wiki.php.net/rfc/phar_stop_autoloading_metadata.

[7] B. A. Azad, P. Laperdrix, and N. Nikiforakis, “Less is More: Quantifying the Security Benefits of Debloating Web Applications,” in *USENIX Security Symposium (SEC)*, 2019, pp. 1697–1714.

[8] A. Bulekov, R. Jahanshahi, and M. Egele, “Saphire: Sandboxing PHP Applications with Tailored System Call Allowlists,” in *USENIX Security Symposium (SEC)*, 2021, pp. 2881–2898.

[9] S. Cao, B. He, X. Sun, Y. Ouyang, C. Zhang, X. Wu, T. Su, L. Bo, B. Li, C. Ma, J. Li, and T. Wei, “ODDFUZZ: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing,” 2023.

[10] G. L. Chris Frohoff, “Marshalling Pickles: How Deserializing Objects Will Ruin Your Day (AppSecCali2015),” https://appseccalifornia2015.sched.com/event/40c922b93ac45988f1be4da3dea27892#.VjpyL36rRhE.

[11] S. Cui, G. Zhao, Z. Dai, L. Wang, R. Huang, and J. Huang, “Pyinfer: Deep learning semantic type inference for python variables,” 2021.

[12] J. Dahse, N. Krein, and T. Holz, “Code Reuse Attacks in PHP: Automated POP Chain Generation,” in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 42–53.

[13] L. Damas and R. Milner, “Principal type-schemes for functional programs,” in *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1982, pp. 207–212.

[14] S. Esser, “Shocking News in PHP Exploitation,” https://owasp.org/www-pdf-archive/POC2009-ShockingNewsInPHPExploitation.pdf.

[15] C. Frohoff, “ysoserial,” https://github.com/frohoff/ysoserial.

[16] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks, “Static Type Inference for Ruby,” in *ACM Symposium on Applied Computing (SAC)*, 2009, pp. 1859–1866.

[17] GitHub, “GitHub search for native deserialization usage,” https://github.com/search?q=language%3Aphp+unserialize%28+OR+serialize%28&type=repositories.

[18] ——, “GitHub security advisory search for deserialization vulnerability,” https://github.com/advisories?query=cwe%3A502 [Accessed 26. Jun. 2023].

[19] B. Hackett and S.-y. Guo, “Fast and Precise Hybrid Type Inference for JavaScript,” *ACM SIGPLAN Notices*, vol. 47, no. 6, pp. 239—250, 2012.

[20] M. Hassan, C. Urban, M. Eilers, and P. Müller, “MaxSMT-Based Type Inference for Python 3,” in *International Conference on Computer Aided Verification (CAV)*, 2018, pp. 12–19.

[21] M. Hills, P. Klint, and J. Vinju, “An Empirical Study of PHP Feature Usage: A Static Analysis Perspective,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 325–335.

[22] F. Inc, “Hacklang,” https://hacklang.org/.

[23] ——, “HHVM,” https://hhvm.com/.

[24] M. P. Inc, “Pyre: A performant type-checker for Python 3,” https://pyre-check.org.

[25] V. Jacobson, “Rfc1144: Compressing tcp/ip headers for low-speed serial links,” USA, 1990.

[26] R. Jahanshahi, B. Amin Azad, N. Nikiforakis, and M. Egele, “Minimalist: Semi-automated Debloating of PHP Web Applications through Static Analysis,” in *USENIX Security Symposium (SEC)*, 2023, pp. 5557–5573.

[27] JetBrains, “PhpStorm - The Lightning-Smart PHP IDE,” https://www.jetbrains.com/phpstorm/.

[28] M. Kazerounian, J. S. Foster, and B. Min, “SimTyper: Sound Type Inference for Ruby Using Type Equality Prediction,” *ACM on Programming Languages*, vol. 5, no. OOPSLA, pp. 1–27, 2021.

[29] M. Kazerounian, B. M. Ren, and J. S. Foster, “Sound, Heuristic Type Annotation Inference for Ruby,” in *ACM SIGPLAN International Symposium on Dynamic Languages (DLS)*, 2020, pp. 112—-125.

[30] S. Klingström and P. Olsson, “Type Inference in PHP using Deep Learning,” *LU-CS-EX*, 2020.

[31] P. S. Library, “Unserializable, but unreachable: Remote Code Execution on vBulletin,” https://www.ambionics.io/blog/vbulletin-unserializable-but-unreachable.

[32] ——, “wordpress-stubs,” https://packagist.org/packages/php-stubs/wordpress-stubs.

[33] E. Maia, N. Moreira, and R. Reis, “A Static Type Inference for Python,” *Workshop on Dynamic Languages and Applications (DYLA)*, vol. 5, no. 1, p. 1, 2012.

[34] MITRE, “CVE-2021-42321,” https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-42321.

[35] ——, “CWE-843: Access of Resource Using Incompatible Type (’Type Confusion’),” https://cwe.mitre.org/data/definitions/843.html.

[36] NIST, “CVE-2020-10189 Detail,” https://nvd.nist.gov/vuln/detail/CVE-2020-10189.

[37] ——, “National Vulnerability Database,” https://nvd.nist.gov/vuln.

[38] Oracle, “Java Object Serialization,” https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/index.html.

[39] ——, “Serialization Filtering,” https://docs.oracle.com/javase/8/docs/technotes/guides/serialization/filters/serialization-filtering.html#addressing-deserialization-vulnerabilities.

[40] OWASP, “OWASP Top Ten,” https://owasp.org/www-project-top-ten/.

[41] S. Park, D. Kim, S. Jana, and S. Son, “FUGIO: Automatic Exploit Generation for PHP Object Injection Vulnerabilities,” in *USENIX Security Symposium (SEC)*, 2022, pp. 197–214.

[42] PHP, “Autoloading Classes,” https://www.php.net/manual/en/language.oop5.autoload.php.

[43] ——, “PHP include,” https://www.php.net/manual/en/function.include.php.

[44] ——, “PHP: json_encode,” https://www.php.net/manual/en/function.json-encode.php.

[45] ——, “PHP: Magic Methods,” https://www.php.net/manual/en/language.oop5.magic.php#object.tostring.

[46] ——, “PHP Namespaces,” https://www.php.net/manual/en/language.namespaces.php.

[47] ——, “PHP unserialize,” https://www.php.net/manual/en/function.unserialize.php.

[48] PHP-FIG, “PSR-4: Autoloader,” https://www.php-fig.org/psr/psr-4/.

[49] PHP.Watch, “PHP 8.0: phar:// stream wrapper no longer unserializes meta data automatically,” https://php.watch/versions/8.0/phar-stream-wrapper-unserialize.

[50] Python, “json — JSON encoder and decode,” https://docs.python.org/3/library/json.html.

[51] ——, “pickle — Python object serialization,” https://docs.python.org/3/library/pickle.html.

[52] V. Raychev, M. Vechev, and A. Krause, “Predicting Program Properties from "Big Code",” *ACM SIGPLAN Notices*, pp. 111–124, 2015.

[53] E. Romano, “PHP Object Injection,” https://owasp.org/www-pdf-archive/POC2009-ShockingNewsInPHPExploitation.pdf.

[54] V. Saenger, “How To Save And Share your Machine Learning Models (Plus More), All With One File,” https://towardsdatascience.com/how-to-save-and-share-your-machine-learning-models-plus-more-all-with-one-file-a2536dd38883.

[55] I. Sayar, A. Bartel, E. Bodden, and Y. Le Traon, “An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, pp. 1–45, 2022.

[56] A. Security, “PHPGGC: PHP Generic Gadget Chains,” https://github.com/ambionics/phpggc.

[57] H. Shacham, “The Geometry of Innocent Flesh on the Bone: Return-into-Libc without Function Calls (on the X86),” in *ACM Conference on Computer and Communications Security (CCS)*, 2007, pp. 552—-561.

[58] E. Sultanik, “Never a dill moment: Exploiting machine learning pickle files,” https://blog.trailofbits.com/2021/03/15/never-a-dill-moment-exploiting-machine-learning-pickle-files/.

[59] M. D. S. R. Team, “Understanding type confusion vulnerabilities: CVE-2015-0336,” https://www.microsoft.com/en-us/security/blog/2015/06/17/understanding-type-confusion-vulnerabilities-cve-2015-0336/.

[60] Vimeo, “Psalm - a static analysis tool for PHP,” https://psalm.dev.

[61] VKCOM, “noverify - Pretty fast linter (code static analysis utility) for PHP,” https://github.com/VKCOM/noverify.

[62] M. Weiser, “Program slicing,” *IEEE Transactions on Software Engineering (TSE)*, no. 4, pp. 352–357, 1984.

[63] A. Wollrath and K. Bharat, “Pickling State in the Java System,” *USENIX Conference on Object Oriented Technologies and Systems (COOTS)*, pp. 22–32, 1996.

[64] Z. Xu, X. Zhang, L. Chen, K. Pei, and B. Xu, “Python Probabilistic Type Inference with Natural Language Support,” in *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016, pp. 607—-618.

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

[66] H. Zhao, I. Proctor, M. Yang, X. Qi, M. Williams, Q. Gao, G. Ottoni, A. Paroski, S. MacVicar, J. Evans *et al.*, “The HipHop compiler for PHP,” *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 575–586, 2012.