ObjectMap: Detecting Insecure Object Deserialization

Nikolaos Koutroumpouchos
nikoskoutr@ssl-unipi.gr
Department of Digital Systems,
University of Piraeus, Piraeus Greece

Georgios Lavdanis
georlav@ssl-unipi.gr
Department of Digital Systems,
University of Piraeus, Piraeus Greece

Eleni Veroni
veroni@unipi.gr
Department of Digital Systems,
University of Piraeus, Piraeus Greece

Christoforos Ntantogian
dadoyan@unipi.gr
Department of Digital Systems,
University of Piraeus, Piraeus Greece

Christos Xenakis
xenakis@unipi.gr
Department of Digital Systems,
University of Piraeus, Piraeus Greece

ABSTRACT

In recent years there is a surge of serialization-based vulnerabilities in web applications which have led to serious incidents, exposing private data of millions of individuals. Although there have been some efforts in addressing this problem, there is still no unified solution that is able to detect implementation-agnostic vulnerabilities. We aim to fill this gap by proposing ObjectMap, an extendable tool for the detection of deserialization and object injection vulnerabilities in Java and PHP based web applications. Furthermore, we also introduce the first deserialization test environment which can be used to test deserialization vulnerability detection tools and for educational purposes. Both of these tools are easily extendable and the first to implement this combination of features to the best of our knowledge and they bring together a synthesis of cross-complementing functionalities that are able to ignite further research in the field and help in the development of more feature-rich solutions.

CCS CONCEPTS

• **Security and privacy → Vulnerability scanners**; **Web application security**; *Penetration testing*.

KEYWORDS

insecure deserialization, web application, security, vulnerability scanner

ACM Reference Format:
Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: Detecting Insecure Object Deserialization. In *Nicosia '19: PCI 2019 - 23rd Panhellenic Conference on Informatics, November 28–30, 2019, Nicosia, CY*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Insecure deserialization is a vulnerability that occurs when untrusted data are deserialized and used to abuse the application logic, inflict denial of service (DoS) attacks, or even execute arbitrary code. This class of vulnerabilities is included in the ten most critical web application security risks of OWASP [31]. In order to understand what insecure deserialization is, we first must understand what the serialization and deserialization functionalities are. Complex modern systems are highly distributed, as the components communicate with each other and share information (such as moving data between services, storing information, etc.), the native binary format is not ideal for transmission. Serialization, also known as marshaling, refers to a process of converting a native binary object into a format that can be easily stored (for example saved to a file or a database), sent through data streams (for example stdout), or sent over a network. The format in which an object is serialized into, can either be binary or structured text (such as XML, JSON, YAML, etc.) with JSON and XML being two of the most commonly used serialization formats within web applications. On the other hand, deserialization is the exact opposite of serialization, that is, transforming serialized data coming from a file, stream or network socket back to an object identical to the one that the deserialized data came from. Serialization operations are extremely common in architectures that include APIs, microservices, and client-side MVC (Model View Controller). Web applications make use of serialization and deserialization regularly and most programming languages even provide native features to serialize data (especially into common formats like JSON and XML). This process is safe as long as the data and objects used, come from trusted and bug-free sources. Using data from any other provenance poses the risk of a malfunction or a deserialization attack since the incoming serialized data could potentially conceal malicious instructions that will force the deserializing program to execute them. It is frequently possible for an attacker to abuse these deserialization features when the application is deserializing untrusted data that the attacker controls. Successful insecure deserialization attacks could allow an attacker to carry out denial-of-service (DoS) attacks, authentication bypasses and remote code execution attacks. [1] Deserialization attack occurrences are abundant both in the past and in recent years, as can be seen in Table 1 [16], where seven documented CVEs related to insecure deserialization are presented, the most recent one was in 2019. It should be noted that the large scale data breach that happened to Equifax in 2017 rooted in insecure deserialization

PCI '19, November 28–30, 2019, Nicosia, CY

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

of the struts framework. [14] Furthermore, in [9], the researchers identified insecure deserialization as a core threat to smart grid systems due to their nature, which requires the transmission of data between nodes that should be serialized and then unserialized in their destination, thus providing a larger than usual attack surface.

In [24], the researchers have found two new previously unknown high severity vulnerabilities in Android related to Java object deserialization. The first is in the Android Platform and the second in Google Play Services. The Android platform vulnerability affects Android 4.3-5.1, M (Preview 1), that is 55% of Android devices at the time of the research writing. This vulnerability allows for arbitrary code execution in the context of many apps and services and also results in privilege escalation. The researchers also demonstrated a Proof-of-Concept exploit against the Google Nexus 5 device, that achieves code execution inside the highly privileged system_server process, and then either replaces an existing arbitrary application on the device with a malware application or changes the device's SELinux policy which would allow the easier exploitation of the device. The researchers also exploited other devices, that they were able to gain kernel code execution by loading an arbitrary kernel module. The Android security team tagged this vulnerability with the CVE-2015-3825 (internally as ANDROID-21437603/21583894) and patched Android 4.4 / 5.x / M and Google Play Services.

Deserialization attacks, despite Java, also affect other languages such as PHP and Python. A PHP Object Injection vulnerability occurs when not sanitized input is used during the deserialization of data in a given web application. The PHP functionalities serialization and deserialization that allow for data storage of any type in a simple string. This format makes it easy to transfer complicated data structures and is often misused to create multidimensional cookies and similar data structures. Since PHP allows deserialization of arbitrary objects, an attacker might be able to inject a specially prepared object with an arbitrary set of properties into the application. Depending on the application implementation, an attacker could trigger internal PHP magic functions which in turn could lead to several vulnerabilities such as code injection, SQL injection, path traversal and application denial of service, depending on the context. [8] Furthermore, python is also vulnerable to these same exploitations through its deserialization functionality. More specifically, an attacker that can control the input to a deserialization python function (pickle.loads(serialized_data)) can forge serialized data that will force the system to run any arbitrary code in the context of the web application. [21] In order for a PHP object injection to be possible, the following conditions must be met [27]:

- The web application must include a class that implements an internal PHP magic method which can be used as a gadget in the attack or to begin a property-oriented programming (POP) chain.
- All of the classes used in the attack must be already declared when the target deserialization function is called, otherwise these classes must be supported by object autoloading.

Object injection in PHP is quite common and many recent vulnerabilities have affected large scale applications. More specifically, both WordPress (CVE-2018-20148) and Drupal (CVE-2019-6338) were affected by object injection in vulnerabilities that allowed attackers to execute code, manipulate files and privilege escalation. Other web applications affected include PHPMailer (CVE-2018-19296), Alienvault (CVE-2016-8580) and OpenPSA2 (CVE-2018-1000525).

In 2009, Esser showed that code reuse attacks based on insecure deserialization (similar to return-oriented programming) are viable in PHP-based web applications [10, 11]. More specifically, he introduced an exploitation approach for object injection vulnerabilities in web applications that abuses the ability of an attacker to arbitrarily modify the properties of an object that is injected into a given web application through deserialization. Thus, the data and control flow of the application can be manipulated depending on the injected objects and the term Property-Oriented Programming (POP) was first used. In the past years, many object injection vulnerabilities were detected in popular open-source PHP content management systems such as Wordpress, Drupal, Joomla, and Piwik. They can lead to critical security vulnerabilities, such as remote code execution, and affect a majority of web servers since PHP is the most popular scripting language on the Web with a market share of more than 80% [25].

1.1 General Rules for Prevention

There are many possible prevention routes that a developer might choose when it comes to insecure deserialization protection. The first and most obvious is allowing only authenticated users and processes to have access to the web application, and thus to minimize the chances of the system falling prey to such an exploit, this does not solve the issue entirely though, because there other attack paths that an intruder might choose. In [22] there is a comprehensive list of possible measures that includes the following:

- Do not accept serialized objects from untrusted sources
- The serialization process needs to be encrypted so that hostile object creation and data tampering cannot run
- Run the deserialization code with limited access permissions
- Strengthen the source code's java.io.objectinputstream
- Monitoring the serialization process can help catch any malicious code and breach attempts
- Validate user input
- Use a web application firewall that can detect malicious or unauthorized insecure deserialization
- Prevent deserialization of domain objects
- Use non-standard data formats
- Only deserialize signed data

Not all of these solutions can be implemented in every scenario but, with enough awareness of the issue, a strategy can be formulated that will protect the web application from malicious deserialization activity from the internet.

In this work, we propose a new software tool named ObjectMap for the detection of both PHP and Java deserialization vulnerabilities. To the best of our knowledge, it is the first tool of its class that is able to detect vulnerabilities in both of these programming languages and it can be easily extended to also cover other languages such as python and ruby. We analyze the software architecture of the tool and we discuss several implementation choices. We also introduce the first vulnerable test environment for deserialization vulnerabilities. This tool can be used by researchers for testing the effectiveness of new deserialization vulnerability detection tools as well as by tutors for educational purposes. Both of these tools are

Table 1: Recent Deserialization Attacks

CVE	Description
CVE-2019-6503	Remote code execution in Chatopera Java application.
CVE-2019-10068	Remote code execution in Kentico .NET application.
CVE-2018-7489	Remote code execution in systems that include the Java Jackson XML functionality.
CVE-2018-6496, CVE-2018-6497	Cross-site request forgery that stemmed from insecure deserialization.
CVE-2018-19362	Denial of service in the JBoss application server due to a XML Jackson deserialization vulnerability.
CVE-2017-9805	Remote code execution related to Struts handling of XML deserialization. (Equifax Incident)

open source, hosted openly online (ObjectMap [1] and PHP Object Injection Test Environment [2]) for review with the aim of enabling fellow researchers and security analysts to produce new related research and tools. Finally, we discuss peculiarities and inherent limitations related to the exploitation of the deserialization attacks that our tool inherits. All in all the contributions of this paper are:

- The design and implementation of the ObjectMap deserialization vulnerability detection tool.
- The deserialization vulnerability test environment for the assessment of our tool.

In section 2 there is related work to our research, which includes both tools and academic work. In section 3 the ObjectMap tool is presented and analyzed and in section 4 we show the results we had when using ObjectMap with our test environment. Finally, in section 5 we discuss the results of our work and possible future work when it comes to extending our tool.

2 RELATED WORK

As discussed previously, Java-based applications that do not make proper usage of deserialization can be vulnerable to exploitation which has led to the creation of detection tools and has initiated related research. In [16] the authors analyzed a series of Java vulnerabilities, one of which was insecure deserialization. Two of the vulnerabilities use a deserialization sequence to create a custom class loader, which can be used to define a class with high privileges. Another exploit uses deserialization within a custom thread, to have a specific restricted class loaded by the bootstrap class loader. Furthermore, two exploits use serialization to transfer information through side channels undetected. One of these two exploits (CVE-2013-1489) prepares an instance of a system class in a way that would be impossible when running with limited privileges. More specifically, it manipulates the value of a certain private field of that system class, which holds a bytecode representation of a class that will later be defined by triggering a specific call sequence. This is profitable because the system class will define this custom class in a namespace that provides access to restricted classes. An attacker would prepare the instance of that system class before the actual attack. When the exploit code is to be deployed, it only contains the serialized object. Deserialization of the manipulated instance is possible even when running with limited privileges. The second exploit that uses serialization to bypass information hiding uses a custom output stream to leak declared fields of serializable classes, while their instances are about to be written. This allows for manipulating private fields of system classes.

The ysoserial tool [12] is a collection of utilities and property-oriented programming "gadget chains" discovered in common java libraries that can, under the right conditions, exploit Java applications performing unsafe deserialization of objects. The main driver program takes a user-specified command and wraps it in the user-specified gadget chain, then serializes these objects to stdout. When an application with the required gadgets on the classpath unsafely deserializes this data, the chain will automatically be invoked and cause the command to be executed on the application host. Furthermore, the hackUtils [5] project incorporates ysoserial in its penetration testing suite. In another instance of deserialization vulnerability detection, Serianalyzer [4] is a Java static bytecode analyzer that traces native method calls made by methods called during deserialization. The main purpose of this tool is as a research tool to audit code for dangerous behavior during deserialization. Finally, another tool based on ysoserial, Android Java Deserialization Vulnerability Tester aims to find and exploit deserialization vulnerabilities in the Android ecosystem.

Look-ahead object input streams can be used as a mitigation for Java deserialization issues in cases where deserializing untrusted data cannot be avoided. There are several different LAOIS implementations such as SerialKiller [6], ValidatingObjectInputStream [7] and Contrast-rO0 [23] with most of them lacking in their defense against DoS attacks. The most promising solution is the one shipped by default in Java 9, the JEP 290 Serialization Filtering [26]. Configurable process-wide filters are also available in recent updates to Java 6, Java 7, and Java 8. However, even JEP 290 (as currently implemented) is deficient in its defense against DoS attacks. [29]

Similar to well-understood injection vulnerabilities such as cross-site scripting (XSS) [19] and SQL injection (SQLi) [15], PHP object injection (POI) vulnerabilities in a given application can be detected with the help of taint analysis. [8] Broadly speaking, a vulnerability is manifested when untrusted user input reaches a security-sensitive sink [28]. Several analysis frameworks to detect different kinds of injection vulnerabilities were proposed in the last years [3, 18, 32, 33].

Furthermore, another tool used in the detection of POI vulnerabilities is PHPGGC [30] which is a library of unserialize payloads along with a tool to generate them, from a command line or programmatically. When the tool encounters an unserialize on a website, this tool allows for the automatic generation of the payload that will be

[1] https://github.com/georlav/objectmap
[2] https://github.com/georlav/ObjectInjectionPlayground

used to test if the website is vulnerable or not. PHPGGC is considered to be the PHP equivalent of the ysoserial Java tool mentioned above. Another tool used as plugin in the web application security testing software Burp Suite is the "PHP Unserialize Check". This plugin tries to find PHP Object Injection Vulnerabilities by passing serialized objects and testing for errors. It is the most closely related to our solution since it uses error based detection, but it is not as robust as our tool without the ability to detect Java serialization vulnerabilities. Furthermore, this tool is an extension of a commercial application that comes in contrast with our open-source and free solution, available to the community to facilitate in research endeavors.

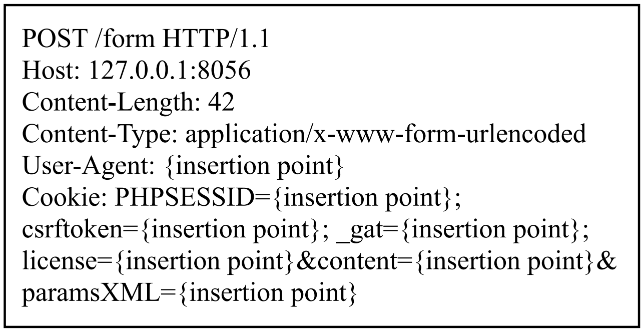
3 OBJECTMAP

The idea was to create a simple command-line tool to help users check web applications developed in PHP or JAVA for insecure deserialization vulnerabilities. The tool is developed in Golang and can be downloaded from https://github.com/georlav/objectmap. The detection is mostly error-based and depends on application error reporting settings. PHP has various options for error reporting and many different error level settings, usually on local development PHP is set to show all kind of errors notices, warnings, errors etc, its also very usual to find applications at production environments with error reporting fully enabled or partially enabled, when partially enabled only fatal errors are visible. Having any kind of error reporting on the production server is considered very bad practice and harmful as it can disclosure useful information to an attacker.

In order to identify possible object injection vulnerabilities, we tried sending different types of data to a request parameter that will be used in an unserialize operation within PHP. Sending random alphanumeric values will make the web application return a simple notice for a deserialization error only if notice-level PHP verbosity is enabled. Since the desired vulnerability detection cannot be based on such hit or miss system (most production systems will not have notice-level reporting), we needed something that will surely cause the web application to report an error when our data were used in an unserialize operation. To solve this problem, we generated valid serialized objects and changed them until errors were starting to occur and get reported on the client-side. Depending on the PHP version that the web application was running, different errors were reported with different information and different error codes, our solution checks for all of these indications to identify deserialization errors. Any found errors are reported as possible object injection vulnerabilities, which depending on the PHP code running could be exploited, the actual exploitation of the application is not in the scope of the ObjectMap vulnerability scanner.

3.1 How does it work

The basic idea behind our solution is that ObjectMap will receive as input a target URL and a variety of options, it will validate and analyze the input, from the input it will generate a combination of requests with various insertion points. The insertion point is any point inside a request that a user can inject a payload, it can be a header, a cookie, post or get parameters even the raw body of the request. (Figure 1)



```
POST /form HTTP/1.1
Host: 127.0.0.1:8056
Content-Length: 42
Content-Type: application/x-www-form-urlencoded
User-Agent: {insertion point}
Cookie: PHPSESSID={insertion point};
csrftoken={insertion point}; _gat={insertion point};
license={insertion point}&content={insertion point}&
paramsXML={insertion point}
```

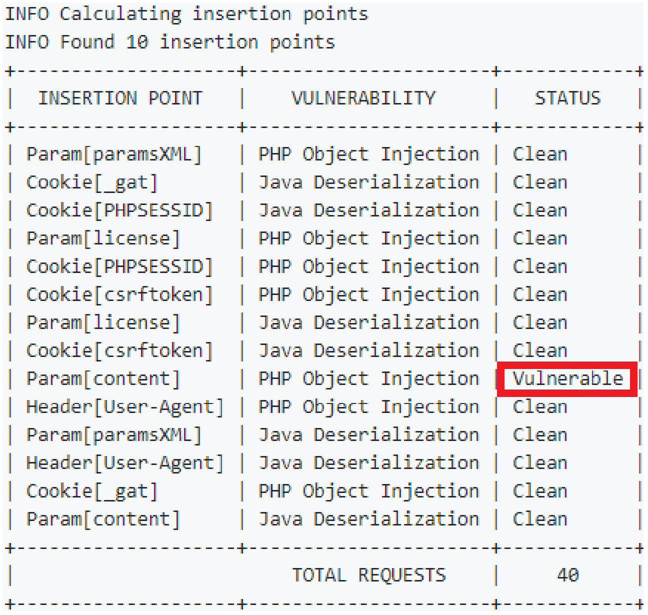
Figure 1: Possible serialized data insertion points in a request.

After identifying all the possible insertion points, ObjectMap will then generate a series of requests that will contain forged payloads injected in each insertion point. These payloads will be designed in a way that will force the target web application to throw an error if the payload is directly passed to a deserialization function on the server-side. This way, our solution will be able to identify any request parameter that possibly exposes the web application to an insecure deserialization vulnerability. The requests generated will be pushed through a shared channel and then several workers (threads) which can be defined by the user, will execute all these requests. The application will then gather all the responses from the target web application and will search for known patterns (deserialization error messages) inside the responses trying to identify if a target is vulnerable, if any possible vulnerabilities are found, then ObjectMap will report from which parts of the request this behavior stemmed. The final report will look like the one shown in Figure 2. This output indicates that for the given request, it found 10 insertion points, for which it generated and executed 40 requests and detected that the paramsXML parameter is possibly vulnerable to PHP object injection.

4 RESULTS AND TEST ENVIRONMENT

For the purposes of the ObjectMap demonstration, we developed the PHP Object Injection playground environment that contains object injection insertion points in various formats. This environment is designed so that we can test all the capabilities of ObjectMap by targeting GET/POST parameters, cookies and the headers of the request, effectively covering the entire client-side attack surface. The playground environment is the first open-source object injection vulnerability testing tool to the best of our knowledge, it is easily extensible and fully dockerized with the aim of helping other researchers and security analysts in their endeavors. For completeness, we executed attacks on the entirety of the request to ensure that our solution was able to successfully inject the desired payload on any of the available positions within the header. More specifically, ObjectMap was able to find all the possible object injection points that existed within our vulnerable website (see Table 2).

In more detail, we first created a custom request that only included HTTP GET parameters and loaded it to ObjectMap. The exact request with the report from ObjectMap can be found in Figure 3. Here the obj POST parameter, found in the first line of the



```
INFO Calculating insertion points
INFO Found 10 insertion points
+-------------------+---------------------+-----------+
|  INSERTION POINT  |    VULNERABILITY    |  STATUS   |
+-------------------+---------------------+-----------+
| Param[paramsXML]  | PHP Object Injection | Clean     |
| Cookie[_gat]      | Java Deserialization | Clean     |
| Cookie[PHPSESSID] | Java Deserialization | Clean     |
| Param[license]    | PHP Object Injection | Clean     |
| Cookie[PHPSESSID] | PHP Object Injection | Clean     |
| Cookie[csrftoken] | PHP Object Injection | Clean     |
| Param[license]    | Java Deserialization | Clean     |
| Cookie[csrftoken] | Java Deserialization | Clean     |
| Param[content]    | PHP Object Injection | Vulnerable |
| Header[User-Agent] | PHP Object Injection | Clean     |
| Param[paramsXML]  | Java Deserialization | Clean     |
| Header[User-Agent] | Java Deserialization | Clean     |
| Cookie[_gat]      | PHP Object Injection | Clean     |
| Param[content]    | Java Deserialization | Clean     |
+-------------------+---------------------+-----------+
|                       TOTAL REQUESTS     |    40     |
+-------------------+---------------------+-----------+
```

Figure 2: A typical result after running ObjectMap against a domain. Here the parameter "Content" is found to be vulnerable.

Table 2: ObjectMap Detection Ability

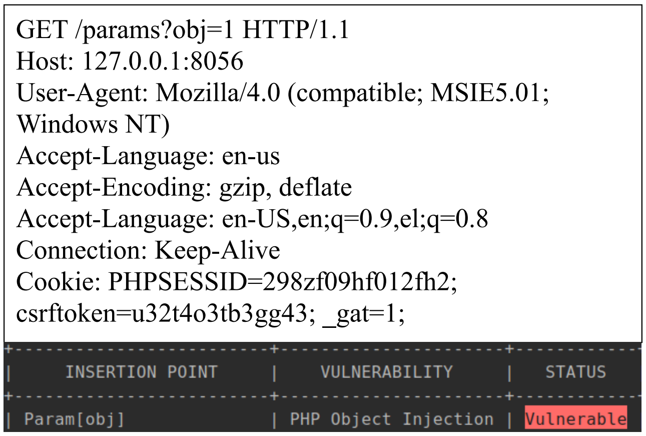
Injection Point	Detectable
GET Parameters	Detected Vulnerability
POST Parameters	Detected Vulnerability
Cookies	Detected Vulnerability
Headers	Detected Vulnerability

request is possibly vulnerable to object injection attacks since it was detected that this parameter is fed directly to an unserialize PHP function. Likewise, in other tests, ObjectMap was able to find POST parameters and Cookie objects to be possibly vulnerable to object injection.

5 DISCUSSION AND FUTURE WORK

ObjectMap's functionality, although powerful, is somehow limited when it comes to robustness and adaptability. A missing feature is the inclusion of a web crawler that would walk through every page and scan not only the request parameters, but also detect input forms that could potentially contain object injection vulnerabilities. This feature would greatly improve the quality of the application as a penetration testing tool by automating the process of detecting object injection vulnerabilities in an entire domain.

Furthermore, this solution could be improved by increasing its impact on other languages. ObjectMap detects deserialization vulnerabilities in Java and PHP, as discussed in section 1 though, these vulnerabilities affect other programming languages such as Python [21] and Ruby [20] which are commonly used in web applications. Our aim is to research how these languages react to bad serialized data and extend ObjectMap so that it can detect possible unsafe deserialization of user-supplied data which could provide exploitation paths.



```
GET /params?obj=1 HTTP/1.1
Host: 127.0.0.1:8056
User-Agent: Mozilla/4.0 (compatible; MSIE5.01;
Windows NT)
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9,el;q=0.8
Connection: Keep-Alive
Cookie: PHPSESSID=298zf09hf012fh2;
csrftoken=u32t4o3tb3gg43; _gat=1;
```

```
+--------------------------+----------------------+--------
|      INSERTION POINT     |    VULNERABILITY     |  STATUS
+--------------------------+----------------------+--------
| Param[obj]               | PHP Object Injection | Vulnerable
```

Figure 3: A vulnerability found in a GET parameter and the result from ObjectMap.

Moreover, the tool aims to only detect possible vulnerabilities without exploiting them. Historically, this specific type of attack was very difficult to successfully execute. This is because creating an automatic exploitation tool for deserialization vulnerabilities requires knowledge of how the target application is actually implemented. In order to exploit such a vulnerability, the attacker needs to go through a trial and error procedure until he can find out if the deserialization function is vulnerable. Although this approach cannot be automated in general, there are two tools that we mentioned before (ysoserial [12], PHPGGC [30]) that can automatically exploit common vulnerabilities that have known exploitation payloads. Of note, these tools can easily be integrated within the ObjectMap workflow to actually test if the found deserialization insertion points are vulnerable to these common exploits.

These tools are limited in their functionality as they depend on specific preconditions for them to produce effective exploitation payloads. More specifically, GGC makes use of a function introduced in PHP 5, the autoload() magic function [17], which unintentionally made exploitation of deserialization vulnerabilities easier. This auto-loading feature was helpful for PHP developers who did not have to manually declare all the source files they required in the corresponding PHP files, while it also allowed for the creation of PHP package managers such as composer. The downside of this function was that it also allowed the easier exploitation of deserialization functions, because they provided the capability to instantiate any new PHP class across the entire application without the need for them to be declared, thereby enabling the easier construction of gadget chains. This exact new functionality is what the PHPGGC tool utilizes to create exploitation payloads, although it is limited by what packages are installed (and thus auto-loadable) within the PHP web application. More specifically, when a web application contains a list of popular PHP packages such as Doctrine, Symfony, Laravel, Yii and Zend-Framework which are known to contain vulnerabilities, PHPGGC can generate gadget chains (with the usage of the autoload function) to achieve remote code execution, arbitrary file writes, and SQL injections. [2]

On the other hand, ysoserial depends on two criteria for a manifestation of a Java deserialization vulnerability: (1) The software must accept and deserialize data from a source that an attacker could manipulate and (2) the existence of "unsafe" classes (gadgets) in the classpath of the application. [13] Ysoserial contains a database of Java frameworks that are known to contain these unsafe classes and asks from the user to provide it with the target framework so that it can generate the appropriate exploitation payload. This way, ysoserial is also limited (like PHPGGC) to the exploitation of a specific list of vulnerable targets. To the best of our knowledge, there are no general-purpose automatic exploitation toolkits which are not limited by what frameworks or packages are installed in the context of the target web application.

Finally, web applications make use of readily available packages to implement specific functionalities. These packages are installed as-is in various ways (composer, npm, requirement-file) on the back-end of the web application and could potentially contain vulnerabilities such as unsafe deserialization. We aim to improve ObjectMap in order to identify what framework is used by the application while searching for possible vulnerable packages and report them at the end of its operation. Furthermore, since the vulnerabilities will be known, ObjectMap could use pre-loaded payloads in order to check if any of the request parameters are used within those vulnerable packages.

6 CONCLUSIONS

In this work, we analyzed the need for a more sound and complete approach when it comes to detecting deserialization vulnerabilities. Although this class of vulnerabilities has produced large-scale attacks with a considerable economic and social impact, there is still a lack when it comes to complete and automatic deserialization vulnerability detection tools and related research. Through our solution, we aim to cover this research need by providing a novel and complete solution (ObjectMap) that can detect deserialization vulnerabilities in the most common implementations. Furthermore, we also created a deserialization vulnerability test-bed, with which we tested our detection tool, which can initialize novel and innovative research. Both of our proposed solutions are open-source and available to the community to support any research or software development aspirations. Through our work, we hope that new attention is brought to this matter with a more detailed approach that is required to defend against this serious and widespread threat.

7 ACKNOWLEDGMENTS

This work was supported by the European Commission, under the FutureTPM, CUREX, INCOGNITO and SECONDO projects; Grant Agreements no. 779391, 826404, 824015 and 823997, respectively.

REFERENCES

[1] 2017. Deserialization of untrusted data. Retrieved August 19, 2019 from https://www.owasp.org/index.php/Deserialization_of_untrusted_data

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. 2019. Less is more: quantifying the security benefits of debloating web applications. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 1697–1714.

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

[4] Moritz Bechler. 2015. Serianalyzer. Retrieved September 1, 2019 from https://github.com/mbechler/serianalyzer

[5] Brianwrf. 2015. hackUtils. Retrieved September 1, 2019 from https://github.com/brianwrf/hackUtils

[6] Luca Carettoni. 2017. SerialKiller. Retrieved September 5, 2019 from https://github.com/ikkisoft/SerialKiller

[7] Apache Commons. 2019. ValidatingObjectInputStream. Retrieved September 5, 2019 from https://github.com/apache/commons-io/blob/master/src/main/java/org/apache/commons/io/serialization/ValidatingObjectInputStream.java

[8] Johannes Dahse, Nikolai Krein, and Thorsten Holz. 2014. Code reuse attacks in php: Automated pop chain generation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 42–53.

[9] Vasudev Dehalwar, Akhtar Kalam, Mohan Lal Kolhe, and Aladin Zayegh. 2017. Review of web-based information security threats in smart grid. In *2017 7th International Conference on Power Systems (ICPS)*. IEEE, 849–853.

[10] Stefan Esser. 2009. Shocking News in PHP Exploitation. *Power of Community (POC)* (2009).

[11] Stefan Esser. 2010. Utilizing code reuse or return oriented programming in PHP applications. *BlackHat USA* 69 (2010).

[12] Chris Frohoff. 2018. ysoserial. Retrieved September 1, 2019 from https://github.com/frohoff/ysoserial

[13] Apostolos Giannakidis. 2018. The Java Deserialization Problem. Retrieved September 3, 2019 from https://www.waratek.com/java-deserialization-problem/

[14] Jason Gillam. 2017. Equifax Breach. Retrieved August 12, 2019 from https://blog.secureideas.com/2017/09/equifax-breach-why-i-am-not-surprised.html

[15] William G Halfond, Jeremy Viegas, Alessandro Orso, et al. 2006. A classification of SQL-injection attacks and countermeasures. In *Proceedings of the IEEE International Symposium on Secure Software Engineering*, Vol. 1. IEEE, 13–15.

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

[17] Michael Hull. 2016. Autoloading Classes In PHP. Retrieved September 3, 2019 from https://resoundingechoes.net/development/autoloading-classes-php/

[18] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. 2006. Pixy: A static analysis tool for detecting web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. IEEE, 6–pp.

[19] Amit Klein. 2002. Cross site scripting explained. *Sanctum White Paper* (2002), 1–7.

[20] John Leyden. 2018. Ruby taken off the rails by deserialization exploit. Retrieved August 19, 2019 from https://portswigger.net/daily-swig/ruby-taken-off-the-rails-by-deserialization-exploit

[21] Dan Lousqui. 2017. Explaining and exploiting deserialization vulnerability with Python. Retrieved September 1, 2019 from https://dan.lousqui.fr/explaining-and-exploiting-deserialization-vulnerability-with-python-en.html

[22] Graeme Messina. 2018. 10 Steps to Avoid Insecure Deserialization. Retrieved September 2, 2019 from https://resources.infosecinstitute.com/10-steps-avoid-insecure-deserialization/

[23] Contrast Security OSS. 2016. Contrast-rO0. Retrieved September 5, 2019 from https://github.com/Contrast-Security-OSS/contrast-rO0

[24] Or Peles and Roee Hay. 2015. One class to rule them all: 0-day deserialization vulnerabilities in android. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*.

[25] Natalya Prokofyeva and Victoria Boltunova. 2017. Analysis and Practical Application of PHP Frameworks in Development of Web Information Systems. *Procedia Computer Science* 104 (12 2017), 51–56. https://doi.org/10.1016/j.procs.2017.01.059

[26] Roger Riggs. 2017. JEP 290: Filter Incoming Serialization Data. Retrieved September 5, 2019 from https://openjdk.java.net/jeps/290

[27] Egidio Romano. 2015. PHP Object Injection. Retrieved August 12, 2019 from https://www.owasp.org/index.php/PHP_Object_Injection

[28] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. 2010. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *2010 IEEE symposium on Security and privacy*. IEEE, 317–331.

[29] Robert Seacord. 2017. Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS).

[30] Ambionics Security. 2019. PHPGGC: PHP Generic Gadget Chains. Retrieved September 1, 2019 from https://github.com/ambionics/phpggc

[31] Andrew van der Stock, Brian Glas, Neil Smithline, and Torsten Gigler. 2018. *OWASP Top 10 -2017*. Technical Report.

[32] Gary Wassermann and Zhendong Su. 2008. Static detection of cross-site scripting vulnerabilities. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. IEEE, 171–180.

[33] Yichen Xie and Alex Aiken. 2006. Static Detection of Security Vulnerabilities in Scripting Languages.. In *USENIX Security Symposium*, Vol. 15. 179–192.