A Composite Discover Method for Gadget Chains in Java Deserialization Vulnerability

Zhaojia Lai[1], Haipeng Qu[1,*] and Lingyun Ying[2]

[1]Ocean University of China, Qingdao, China
[2]QI-ANXIN Techlology Research Institute, Beijing, China

Abstract

The Java deserialization vulnerability is the most dangerous and widely affected. Since this vulnerability was proposed, numerous security practitioners have studied it and developed related detection and defence tools. The discovery of the program's potential gadget chains is the most effective defensive measure. Previously, gadget chains have relied on manual search. Automating discover gadget chains is essential for Java security. However, there are no practical tools to achieve this.

So, we propose a new composite discovery method that generates the corresponding byte streams based on the static analysis results and performs deserialization detection. Our innovation combines serialization protocols and reflection mechanisms to generate objects dynamically and implement attack injection and detection. The evaluation verified its effectiveness, where we found 52 available gadget chains in Apache Commons Collections.

Keywords

Java security, static analysis, dynamic verification

1. Introduction

Java serialization is a mechanism for converting an object to a byte stream, which significantly expands the ability of Java programs to transfer objects in networks[1] and provides the condition for RMI(Remote Method Invocation)[2]. Java deserialization is the reverse of Java serialization. It reconstructs a byte stream to an objects[3]. However, this process could trigger some magic methods that can spontaneously call other methods, perhaps even another magic method. Magic methods make up the gadget chain[4].

An attacker can construct a byte stream to control the method call chain during deserialization and trigger dangerous methods. It could cause a privilege escalation, information disclosure, and RCE(remote code execution)[5]. In addition, this vulnerability is also widely spread. It has dramatically affected many well-known programs such as Weblogic, Jboss, etc[6]. The total of Java deserialization vulnerabilities in CVE(Common Vulnerabilities & Exposures) is increasing yearly. In 2021, up to 17% of Java-related CVEs are related to Java deserialization. Most of such vulnerabilities are high-risk due to RCE(eg.CVE-2021-36981[7], CVE-2021-35464[8]).

Although there are already programs[9, 10, 11] to detect and intercept such attacks, we prefer to detect potential gadget chains during the development process to maintain the security and stability of the software. However, there is no good solution for it.

Gadget Inspector[12] is a tool based on static taint analysis. It uses a very efficient method of symbolic execution, which generates call graphs efficiently. However, the lack of static analysis and search strategies makes it very prone to false positives and negatives. In practice, it is challenging to generate an effective gadget chain.

Rasheed proposed[13] a Fuzzer based on static analysis bootstrap, which guides Fuzzer to generate byte stream through the heap access path. This method relies heavily on the initial results of static analysis. The problem of too few static analysis results and the byte stream's structural variation will lead to unsatisfactory results. Although it will not produce false positives, it will still produce many false negatives.

Therefore, we propose a novel approach to discovering gadget chains. This approach follows the static analysis of Gadget Inspector to obtain the gadget chains to be verified. We still use symbolic execution to generate call graphs in this work. These call graphs can be abstracted into a collection of <caller, callee array>. The gadget Inspector produces false negatives because it uses a breadth-first search algorithm (BFS) to traverse the call graph. This BFS does not consider that multiple gadget chains may share nodes, which leads to only one of the multiple gadget chains passing through the same node will be searched. Therefore, We use a depth-first search algorithm that traverses a single node multiple times to avoid this problem.

To remove the false positives in the static analysis, We propose a matching dynamic verification mechanism

QuASoQ 2022: 10th International Workshop on Quantitative Approaches to Software Quality, December 06, 2022, virtual

* Corresponding author

Redomichelan@stu.ouc.edu.cn (Z. Lai); Quhaipeng@ouc.edu.cn (H. Qu); yinglingyun@qianxin.com (L. Ying)

0000-0002-3402-1438 (Z. Lai)

We propose a matching dynamic verification mechanism based on the Java serialization protocol and reflection mechanisms. This dynamic verification can dynamically generate the object corresponding to each gadget chain. This method, called GCGM(Gadget Core Growth Method), collates a class list based on the gadget chain and generates an object bottom-up according to the gadget core. The implementation of GCGM relies on the Java reflection mechanism, which can get all methods and properties based on a class name. The objects dynamically generated by GCGM will be serialized and injected with malicious behaviour to generate a byte stream, which the deserialization portal can verify *readObject()*.

Our contributions are as follows:

- We have improved the search algorithm of Gadget Inspector to reduce false negatives.
- We propose a dynamic verification to remove false positives based on Java serialization protocol and reflection.

2. Related Work

This section will introduce the gadget chain in Java deserialization vulnerability and some detection efforts. These efforts can be divided into two types, fingerprinting-based detection, and active discovery detection.

2.1. Gadget Chain

The Java deserialization vulnerability and the first gadget chain were discovered and proposed by Chris Frohoff[14]. In the Java deserialization vulnerability, the gadget chain is the method call chain from the deserialization entry method *readObject()* to the command execution method *exec()*[12].

Java deserialization recovers a byte stream into a Java object[3]. This complex refactoring process may trigger some magic methods. A magic method will call another method, even another magic method. For example, *HashMap.put()* is a magic method because it can automatically call *HashMap.hash()*. The successive calls of these magic methods form a gadget chain. The byte stream determines the gadget chain. An attacker can construct a byte stream to control the direction of the gadget chain to trigger some specific methods to achieve remote code execution.

2.2. Fingerprinting-based Detection

Since many security researchers have discovered many gadget chains manually, fingerprinting-based detection implemented by integrating these gadget chains is an efficient detection method.

Ysoserial[14] is a detection program that integrates a large number of gadget chains. It can quickly generate payloads for specific Java libraries to detect Java deserialization vulnerabilities. Marshalsec[15] is a tool similar to Ysoserial, which supports a broader range of libraries but cannot discover gadget chains. The Java Deserialization Scanner[16] can confirm the effectiveness of this strategy. It is a plug-in for the well-known penetration testing tool Burp Suite. It can use Ysoserial to generate payloads for penetration testing of targets for deserialization vulnerabilities.

However, fingerprinting-based detection can only detect the presence of known gadget chains in the program, but not unknown gadget chains in the program.

2.3. Active Discovery Detection

Discovering unknown gadget chains in a program is suitable for software security.

Haken's proposed Gadget Inspector[12] in 2017 is the first to enable the active discovery of gadget chains.

Its implementation relies on the following two key steps:

1. Generate passthrough dataflow and passthrough callgraph using symbolic execution.
2. Search gadget chains in the passthrough callgraph by BFS(Breadth First Search).

Gadget Inspector is an effective tool because it discovers some new gadget chains in the evaluation. However, it produces many false positives and false negatives. False positives are because it is a static analysis tool that does not generate results in the actual deserialization process. False negatives are because its search algorithm does not consider the possibility of multiple gadget chains having common nodes.

In 2020, Rasheed[13] proposed a hybrid analysis strategy to avoid false positives. It uses static analysis results as a guide for fuzzing. The advantage is that it does not make false positives because it will execute a trampoline method and observe if the dynamic sink method is triggered. To get more results, it used fuzzing to mutate the byte stream. However, the byte stream of Java serialization is highly structured, which makes fuzzing challenging to perform effectively. This strategy of hybrid analysis provides new ideas for gadget chain discovery, but from its evaluation, it makes a lot of false negatives.

3. Propose Approach

This section proposes a new active discovery detection strategy to reduce false positives and negatives. It is

implemented in two steps: static analysis and dynamic verification.

Firstly, the static analysis makes many gadget chains. This step is similar to Gadget Inspector, but we optimize the search algorithm to get as many gadget chains as possible to reduce false negatives. The dynamic verification dynamically generates the corresponding byte stream based on each gadget chain. These byte streams trigger the detector during deserialization, while false positives cannot complete this process.

3.1. Gadget Core

Our work relies on the fact that multiple gadget chains in the target program may have common vital nodes. We call such vital nodes *gadget core*. A gadget chain can be abstracted as *source->gadget core->sink*. The subchain *gadget core->sink* is called the *core chain*. The subchain *source->gadget core* is called the *edge chain*.

In a target program, gadget core is always sparse, *core chain* is always unique. So we simplify the discovery of the gadget chain to the discovery of the subchain *edge chain*.

Figure 1 shows a gadget chain of the ACC(Apache Common Collection) library. With the introduction of the gadget core, the search for gadget chain can be simplified to the *edge chain*(HashSet.readObject()->LazyMap.get()), whitch saves a lot of costs.

3.2. Static Analysis

Static analysis is a technique for fast white-box testing. Generally, it includes static tainted analyses and static symbolic execution. Static analysis techniques are commonly used in method call chain searches[17]. The symbolic execution algorithm used by Gadget Inspector[12] is good enough, and we rely on it for our work.

Our static analysis is divided into the following steps:

1. Obtain the class information and method information of the target program.
2. Generate call graph by symbolic execution.
3. Search all the *edge chain*.

In the first stage, all the class information, method information, and inheritance relationships of the target class will be obtained. This work will be implemented by ASM library[18], an excellent Java byte stream manipulation tool.

After that, we use the symbolic execution of the Gadget Inspector to obtain the call relationship for every method. These call relationships make up the call graph. This call graph is stored as a collection of <caller, callee array>.

In the last step, we want the search algorithm to discover all the *edge chain*. Figure 2 shows a typical call graph with four gadget chains. Gadget Inspector can only find two chains because it can only visit **E** and **F** once.

Therefore, we propose a DFS(Depth First Search).

This DFS has two key parameters. One is the MTV(maximum time of visits) per node, and the other is the MCL(maximum chain length) in the search. The MTV setting allows DFS to search as many gadget chains as possible by visiting a node multiple times in a search. MCL limits the search depth, preventing DFS from searching too long and meaningless gadget chains. It backtracks when loops are encountered, when chain lengths exceed limits and when the visit times to a node exceed the limit. The DFS keeps a temporary chain in the search, saves the temporary chain to the result, and backtracks when the search reaches the sink method (gadget core).

This strategy ensures we get as many results as possible without timeouts or memory overflows.

3.3. Dynamic Verification

In this work, we dynamically generate byte streams and deserialize them to verify availability based on each gadget chain. Our work relies on the Java serialization protocol[19] and the Java reflection[20].

3.3.1. Gadget Core Growth Method

Java reflection allows the program to load the **Class** object based on a class name[20]. The **Class** object contains the metadata of the class, including all the methods and properties. This mechanism and the proposal of the gadget core led to the design of GCGM (Gadget Core Growth Method).

The GCGM will generate an object for each *edge chain* based on the static analysis results, which works as follows:

1. Get the manually constructed gadget core object as **current object** according to the gadget chain, which does not contain malicious behaviour.
2. Get the class name of the node above **current object** in the gadget chain as **clazz**.
3. Use reflection to get the constructor of **clazz**.
4. Call the constructor of **clazz** with the **current object** as an argument to construct a **new object**.
5. Set **new object** to **current object**.
6. Repeat steps 2-5.

The pseudocode for this method is shown in Algorithm1:

3.3.2. Deserialization Verification

The GCGM allows us to generate objects dynamically based on an *edge chain*. The ultimate goal of dynamic

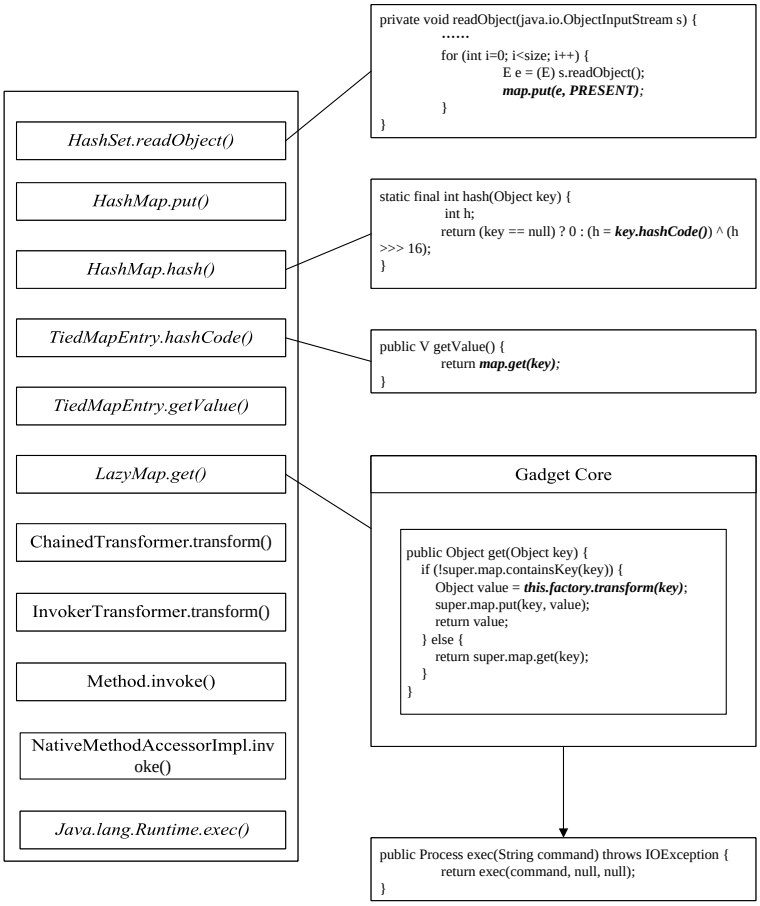

Figure 1: A classic gadget chain in ACC. *LazyMap.get()* could be a gadget core.

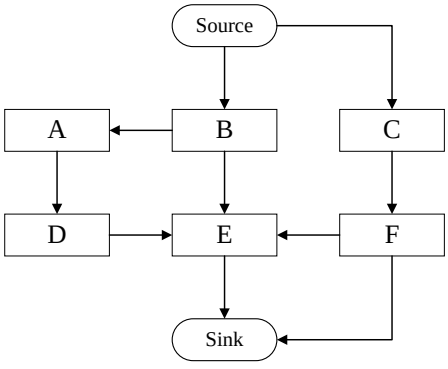

Figure 2: A typical call graph.

verification is to determine whether the corresponding byte stream of this object can trigger the sink method during the deserialization process. We design a unique verification method based on the serialization protocol in this work. This work is based on the highly structured nature of serialized byte stream.

Figure 3 shows the comparison of two serialized byte streams. The left of the image shows the byte stream corresponding to a gadget chain, and the right shows the byte stream corresponding to the corresponding *edge chain*. The difference is the byte stream corresponding to the *core chain*. This allows us to modify the byte stream to add incomplete *edge chain* to the gadget chain.

Figure 4 illustrates our workflow, which has the following steps.

1. Get the results of static analysis, preprocessing each *edge chain*.
2. Pass the *edge chain* into GCGM as the parameter to generate the corresponding *edge object*.

Algorithm 1 Gadget Core Growth Mehtod

Input: x_0: an *edge chain* from static analysis; $Core$: a core object of gadget chain parser, which contains information about a gadget core.

Return: an object generated by GCGM;

set $CurrentObject$ = Core.getCoreObject(x_0);
set $ClassList$ = Core.getClassList(x_0);
for i = 0; i<$ClassList$.length();i++ **do**
    $CurrentClassName$ = $ClassList[i]$;
    $Constructor$ = $Reflection$.getConstrctor($CurrentClassName$);
    $Fields$ = $Reflection$.getFilds($CurrentClassName$);
    $Types$ = Reflection.getClass($CurrentObject$) + Reflection.getInterface($CurrentObject$)
    for $Field$ in $Fields$ **do**
        for $Type$ in $Types$ **do**
            if $Filed$=$Type$ **then**
                $NewObject$ = $Constructor$.newInstance();
                set $NewObject.Field$ = $CurrentObject$;
                $CurrentObject$ = $NewObject$;
            end if
        end for
    end for
end forreturn $CurrentObject$;

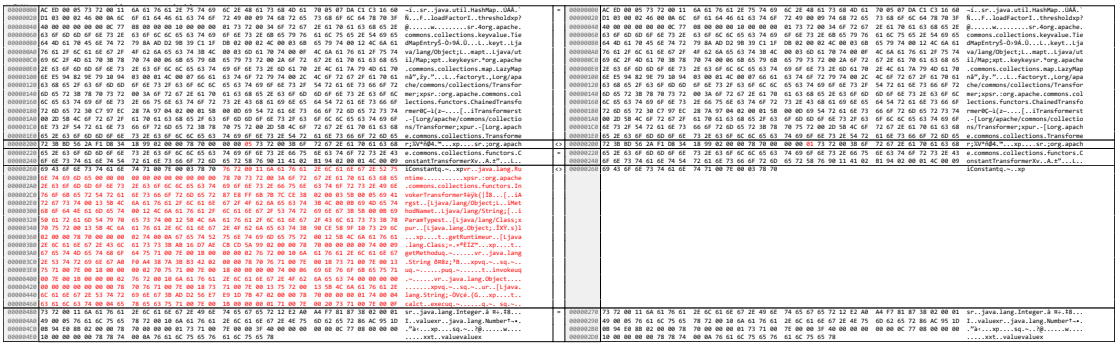

Figure 3: A comparison between the serialized byte stream of two similar objects.

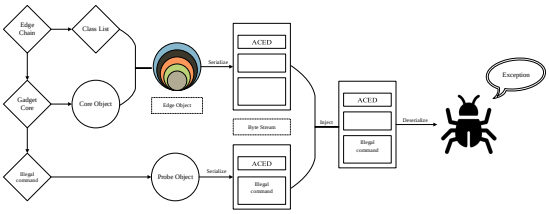

Figure 4: The Process in Dynamic Validation.

3. Generate the *probe object* by passing the *core chain* as an argument. This *probe object* contains an **illegal command** that cannot be executed by Runtime.exec().
4. Serialize the *edge object* and the *probe object* into byte stream.
5. Inject the **illegal command** of the *probe object* into the byte stream of the *edge object* by modifying the byte stream.
6. The acquired byte stream is deserialized, the **illegal command** will be triggered, and the detector will catch a specific exception.

4. Evaluate

To evaluate the effectiveness of our method, we designed feasibility, inefficiency, comparison, and versatility ex-

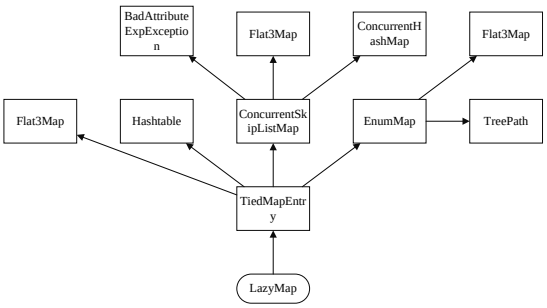

Figure 5: A Growth Tree in ACC,including 7 gadget chains

periments.

Experimental Environment: The experiments were implemented on an Intel(R) Core(TM) i3-10100 CPU @ 3.60GHz with 16GB of RAM on Windows 10.21H1. Gadget Inspector(DFS) and Gadget Catcher were run in Java 8 (release JDK 8u302).

Experimental Setup: In the feasibility experiment, we will execute the method in the test set and determine the availability of the technique based on the results. In the inefficiency experiment, we will observe the impact of two critical parameters of the method on the efficiency and results by modifying these two parameters. In the comparison experiment, we will compare the discovery of our tool with some other methods mentioned so far. In the versatility experiment, we will make discoveries on some other libraries.

Test Set: The following libraries will be used for the test set in this section.

- Apache Commons Collections 3.1
- Commons Beanutils 1.92
- Apache Commons Collections4 4.0
- Jython Standalone 2.5.2.

4.1. Results and Discussion

Feasibility Experiment: LazyMap will be set as a gadget core in ACC. With default settings, a total of 52 gadget chains were discovered. Our results not only hit all three chains in *ysoserial* that are suitable for the JDK version of *CommonsCollections5*, *CommonsCollections6*, and *CommonsCollections7* but also found many other gadget chains, which fully verified the correctness of our strategy. A growth tree is made with some discovered results in Figure 5, the root node is the gadget core, and the page node is the outer class capable of triggering the *readObject()* methods.

Efficiency Experiment: In this experiment, we verify MTV(the maximum times of visits to a node) to optimize default settings.

Table 1
The Result of Efficiency Evaluation Experiment

MTV	Static Analysis		Dynamic Verification	
	Time Cost(s)	Result	Time Cost(s)	Result
50000	862	365724	3673	52
10000	386	74844	125	28
5000	183	37626	58	24
1000	87	7567	8	13
500	51	3816	<1	11
Null	> 3d	/	/	/

Table 2
The Result of Camparion Experiment

Discovery Strategy	Results	Valid Results	Time Cost(min)
Gadget Inspector	4	0	<2
Hybrid Analysis	1	1	<20
Composite Discover	52	52	>60

Table 1 shows the results of the efficiency experiments. This result shows that the number of experimental results and the time cost are positively related to MTV. At an MTV of 50,000, it is possible to obtain over 360,000 results in static analysis and 52 gadget chains after dynamic calibration, when the time spent is about 1 hour. The rule that can be summarized is that when MTV is set more significant, more results can be obtained, but the time overhead is also more; when MTV is infinite, the results cannot be obtained in the expected time.

Comparison Experiment:

In this experiment, we use the ACC library as the test set. Table 2 shows the three strategies' search results and the valid results, Where the results of Gadget Inspector are from our experiments. The experimental results of the hybrid analysis strategy are from the original article. It shows the effectiveness of our discovery strategy over the other two discovery strategies.

In addition, we also performed a simple runtime comparison. Gadget Inspector, a static analysis tool, was able to produce results in two minutes, the hybrid analysis approach was able to get results in 19 minutes, while our method took more than 1h when the MTV was set to 50,000.

Versatility Experiments: We are also trying to use our strategy for gadget chain discovery for some other libraries. In this experiment, we found a new gadget chain in ACC4 by combining the static analysis results shown in Figure 6. This new gadget chain is not currently included in *ysesorial* and can be judged as a new gadget chain. In addition, the *TransformingComparator()* method in this gadget chain can be used as a new gadget core to implement the discovery of other types of gadget chains. In our experiments, we initially verified the feasibility

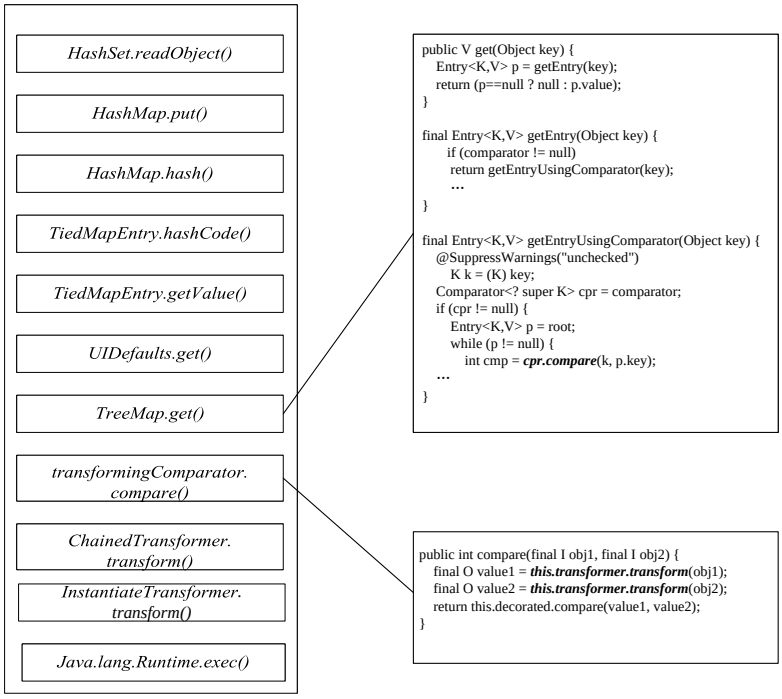

Figure 6: A new gadget chain with a new path to trigger gadget core.

Table 3
General experiments

library	release	gadget core
commons-beanutils	1.9.2	BeanComparator
commons-collections4	4.0	TransformingComparator
jython-standalone	2.5.2	Comparator

experiments of the three libraries in Table 3 and proved that this gadget core could be applied to discover gadget chains.

5. Conclusion

Based on the previous work, we have completed our analysis strategy. This strategy overcomes the common false positives and negatives in gadget chain discovery. The experimental results also prove the correctness and efficiency of our design. We also have a massive advantage in comparing with other strategies.

On the other hand, this strategy also has limitations that rely on manual analysis. Finding a suitable gadget core and building its validation strategy is necessary before analyzing a new library.

References

[1] T. Greanier, Discover the secrets of the java serialization api, 2021. URL: https://www.oracle.com/technical-resources/articles/java/serializationapi.html.

[2] Docs.Oracle.com, Trail: Rmi (the java™ tutorials), 2020. URL: https://docs.oracle.com/javase/tutorial/rmi/.

[3] Docs.Oracle.com, Objectinputstream (java platform se 7), 2011. URL: https://docs.oracle.com/javase/7/docs/api/java/io/ObjectInputStream.html.

[4] M. Daconta, When runtime.exec() won't, 2000. URL: https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities.pdf.

[5] J. Forshaw, Are you my type?, 2012. URL: https://media.blackhat.com/bh-us-12/Briefings/Forshaw/BH_US_12_Forshaw_Are_You_My_Type_WP.pdf.

[6] B. Stephen, What do weblogic, websphere, jboss, jenkins, opennms, and your application have in common? this vulnerability., 2015. URL: https://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability.

[7] C. D. 2021, Cve-2021-36981(vulnerability in sernet

verinice 1.22.2), 2021. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-36981.

[8] C. D. 2021, Cve-2021-35464(vulnerability in forgerock am server 7.0), 2021. URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3546.

[9] L. Carettoni, Defending against java deserialization vulnerabilities, 2016. URL: https://www.ikkisoft.com/stuff/Defending_against_Java_Deserialization_Vulnerabilities.pdf.

[10] S. Cristalli, E. Vignati, D. Bruschi, A. Lanzi, Trusted execution path for protecting java applications against deserialization of untrusted data, in: International Symposium on Research in Attacks, Intrusions, and Defenses, Springer, 2018, pp. 445–464.

[11] N. Koutroumpouchos, G. Lavdanis, E. Veroni, C. Ntantogian, C. Xenakis, Objectmap: Detecting insecure object deserialization, in: Proceedings of the 23rd Pan-Hellenic Conference on Informatics, 2019, pp. 67–72.

[12] I. Haken, Automated discovery of deserialization gadget chains, Proceedings of the Black Hat USA (2018).

[13] S. Rasheed, J. Dietrich, A hybrid analysis to detect java serialisation vulnerabilities, in: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1209–1213.

[14] C. Frohoff, G. Lawrence, ysoserial (a proof-of-concept tool for generating payloads that exploit unsafe java object deserialization.), 2015. URL: https://github.com/frohoff/ysoserial.

[15] M. Bechler, marshalsec, 2017. URL: https://github.com/mbechler/marshalsec.

[16] F. Dotta, Reliable discovery and exploitation of java deserialization vulnerabilities, 2017. URL: https://techblog.mediaservice.net/2017/05/reliable-discovery-and-exploitation-of-java-deserialization-vulnerabilities.

[17] Y. Li, T. Tan, Y. Zhang, J. Xue, Program tailoring: Slicing by sequential criteria, in: 30th European Conference on Object-Oriented Programming (ECOOP 2016), Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[18] E. Bruneton, R. Lenglet, T. Coupaye, Asm: a code manipulation tool to implement adaptable systems, Adaptable and extensible component systems 30 (2002).

[19] Docs.oracle.com, Java serialization protocol, 2014. URL: https://docs.oracle.com/javase/8/docs/platform/serialization/spec/protocol.html.

[20] Docs.oracle.com, Java reflection api, 2014. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/index.html.