Analyzing Prerequisites of known Deserialization Vulnerabilities on Java Applications

Bruno Kreyßig
Umeå University
Umeå, Sweden
bruno.kreyssig@cs.umu.se

Alexandre Bartel
Umeå University
Umeå, Sweden
alexandre.bartel@cs.umu.se

ABSTRACT

We analyze known deserialization exploits targeting applications developed in the Java programming language. As previous research implies, fully comprehending this type of vulnerability is no easy task due to the complexity of exploitation, mostly relying on so-called gadget chains. Even considering known gadget chains, knowledge about their prerequisites is rather limited. In particular, the full range of external library versions, adding exploitable gadgets to the Java classpath was formerly only partially examined. We contribute an in-depth analysis of publicly available Java deserialization vulnerabilities. Specifically, we experimentally assess the prerequisites for exploitation, using 46 different gadget chains on 244 JDK and 5,455 Java dependency versions. Previous research only covered 19 of these gadget chains. Furthermore, we develop a command line tool, *Gadgecy*, for lightweight detection of whether a given Java project contains dependency combinations that enable gadget chains. Using this tool, we conduct an analysis of 2,211 projects from the Apache Distribution directory and 400 well-known Github repositories. The outcome reveals that (1) deserialization exploits apply to recent JDK and library versions, (2) these gadget chains are not being fully reported, and (3) are frequently present in popular Java projects (such as *Apache Kafka* or *Hadoop*).

CCS CONCEPTS

• **Security and privacy → Software and application security**; • **Software and its engineering → Software defect analysis**; *Object oriented development*.

KEYWORDS

serialization, deserialization, gadget chain, Java, vulnerabilitiy, dependency

1 INTRODUCTION

Whenever passing data between systems, serialization protocols are inherently being used. Given the rising popularity of distributed software architectures, the demand for keeping the serialization interface secure also increases. Ranked as one of the *OWASP Top Ten* vulnerabilities [26], insecure deserialization is renowned as a serious security flaw within software applications. This kind of vulnerability stems from the assumption that serialized data being transferred between systems is trusted, and indeed the attack model in general implies that the attacker has full control over the supplied data. In contrast to injection-type vulnerabilities, however, the Java serialization protocol is too complex to rely on a whitelisting mechanism [19]. Previous research has discovered, that it is difficult to pinpoint the vulnerability to a specific section of code. Rather, it is linked to the unfortunate constellation of multiple method invocations that can be combined into so-called *gadget chains* [19, 32, 42]. As discovered in previous research, Within the security community, the popular *Ysoserial* tool [29] is an established repository for validated gadget chains that lead to the exploitation of Java applications. The impact of these exploits ranges from remote code execution (RCE) to arbitrary file uploads and network-based attacks.

From a developer's perspective, it would prove helpful to know whether a given application is susceptible to insecure deserialization. One approach consists of extracting all potentially dangerous gadget chains within the source code and the libraries present on the classpath. Research towards an algorithm achieving this goal of detecting all gadget chains (known and unknown) is currently being worked on [17–19, 32, 40], but has yet to show promising results in terms of completeness and computational overhead. Limitations in dealing with the path explosion problem in Java runtime polymorphism, and means of validation in subsequent property oriented programming must still be overcome. Even then, the algorithm is likely to be too resource-intensive to be executed within each cycle of a software development process. This implies the demand for a lightweight alternative for detecting whether an application is susceptible to a publicly known gadget chain (e.g., from the *Ysoserial* repository) through analysis of software properties (JDK build version and external dependencies).

The National Vulnerability Database (NVD) lists a range of CVEs capturing deserialization vulnerabilities. In October 2023, 1,067 records match the corresponding CWE-502 (*Common Weakness Enumeration*) [38]. Albeit capable of mapping Java library CPEs[1] to CVEs on a unary scale (CVE-2015-6420 perfectly maps the *CommonsCollections* payloads from *Ysoserial* to the affected library versions [39]), the reporting format in particular fails to acknowledge exploits relying on gadgets originating from multiple libraries. Arguably, the CPE format definition is not capable of this task. Furthermore, the 1,067 CVE entries refer, in general, to vulnerabilities in software products built for the Java execution environment. Therefore, it is not possible to extract information regarding the root cause of the vulnerability – JDK and library versions – from CVEs. In the previous work from Sayar et al. [42], an experiment was conducted in order to analyze the library versions and JDKs affected by 19 RCE vulnerabilities within the *Ysoserial* Repository. The research goal was hereby to figure out the specific update that introduced the gadget, and the patch, which later remediated the gadget. As a byproduct, it became clear that the amount of library versions, which enables insecure deserialization, is in fact much higher than

[1]The *Common Platform Enumeration* is used alongside the CVE to uniquely identify soft- and hardware configurations to which a given CVE applies.

the versions mentioned within *Ysoserial*. We use this as a starting point for defining our research questions:

> **RQ1:** What are the dependency and JDK prerequisites for known gadget chains?

To gain a full picture of the prerequisites for the exploitation of Java deserialization vulnerabilities, we conduct a large-scale analysis of 46 gadget chains from the main branch of the *Ysoserial* repository, the *newgadget* branch, unresolved pull requests, and the separate CVE-2022-36944 PoC [46] on the combination of **244** JDK and **5,455** Java library JAR files. The results show that gadget chains are present in recent versions of libraries and JDK releases.

> **RQ2:** How are gadget chains being disclosed?

Through the given mapping of CVE identifiers to library versions within the Maven repository, we find that the CVE discloses only **14** of **46** gadget chains. We also discuss why we believe the CVE is not an appropriate solution for reporting dependencies used in gadget chains and provide an alternate solution through the development of the tool *Gadgecy*.

> **RQ3:** How to determine whether a given Java application contains the dependency prerequisites for insecure deserialization?

Given that the majority of gadget chains rely on external dependencies, we build *Gadgecy* to analyze Java projects and *POM* (*Project Object Model*) dependency files. We then evaluate this tool on 2,211 popular Java-based projects in the Apache Distribution directory [1] and 400 Github Java projects. As a result, **10,76%** and **9,12%** of the projects within the respective evaluation data sets contain the dependency (versions) required for a known gadget chain.

The remaining of the paper is organized as follows. In Section 2, we give the technical background to deserialization vulnerabilities, and analyze potential impacts. Section 3 lays out the experiment framework. To answer our research questions, we evaluate the results in Section 4. Finally, we conclude with a brief review of related contributions, limitations, and a summary in Sections 5, 6 and 7.

2 BACKGROUND

The exploitation of deserialization vulnerabilities aims at making use of flaws in the way a target application handles reconstructing objects from serialized input. In a trivial sense, this could mean an attacker directly controlling fields within the serialized data that affect the application behavior, e.g., setting an *isAdmin*-value from false to true. While this example technically can also be described as insecure deserialization, when it comes to Java deserialization vulnerabilities, the biggest concern originates from supplying seemingly unrelated objects. Upon invocation of the deserialization process (in general, the *readObject()* method; see Listing 1), these objects exploit weaknesses in the Java execution environment rather than relying on the core application logic. Furthermore, this invocation is triggered before the serialized object being read is cast to its intended object type [32].

```
1  ObjectInputStream ois = new ObjectInputStream(
2      new FileInputStream("in.ser"));
3  MyObject o = (MyObject) ois.readObject();
```

Listing 1: Deserialization entry point

Take the following example of the *URLDNS* gadget chain, illustrated in Listings 2, 3 and 4. For reasons of interoperability, when objects are being passed between different JVMs, the *HashMap* class overrides the default behavior of the *readObject()* method. While seemingly insignificant, thereby the *hashCode()* method of an arbitrary object is called (see Listing 2, lines 6 and 11). Recall that the attacker has full control over the serialized data and thus also the object fields. This implies the attacker may now supply any object overriding *hashCode()* to invoke the object-specific implementation of the method. This method may again contain interesting method calls. Hence comes the notion of gadgets and gadget chains.

```
1  private void readObject(ObjectInputStream s) {
2  /* l. 1517-1555 omitted */
3  for (int i = 0; i < mappings; i++) {
4      K key = (K) s.readObject();
5      V value = (V) s.readObject();
6      putVal(hash(key), key, value, false, false);
7  }}
8  static final int hash(Object key) {
9      int h;
10     return (key == null) ? 0 :
11         (h = key.hashCode()) ^ (h >>> 16);
12 }
```

Listing 2: URLDNS gadgets in HashMap [5]

A **gadget** describes a method that opens up further method invocations in additional gadgets. Notice that these **gadgets** may originate from any class provided by the entire classpath. Accordingly, a **gadget chain** combines multiple gadgets in such a way, that an execution path from a serialization entry point to a security sensitive method (i.e., sink) is reached. The sink method determines the impact of the gadget chain. For example, this could be *Runtime.getRuntime().exec()* leading to remote code execution or in the *URLDNS* example *InetAddress.getByName()* (Listing 3, line 8) performing an arbitrary DNS request. It follows that exploitable gadget chains are generally not the result of a single programming weakness, but rather the combination of all gadgets in the gadget chain. This makes gadget chains hard to detect and remediate [42]. Taking these definitions into account, a **Java deserialization vulnerability** exists only if both an exploitable gadget chain can be derived from the application classpath and the application exposes an insufficiently hardened deserialization entry point.

Continuing with the example, a *URL* object within the *HashMap* would delegate the invocation of *hashcode()* to a *URLStreamHandler* (see Listing 3). Finally, the *URLStreamHandler* calls *getHostAdress()* (see Listing 4, line 3), which then continues to the URLs *getHostAddress()* method (line 7) and resolves the host's domain name (Listing 3, line 8). In doing so, this invokes a DNS request to look up the

Table 1: URLDNS gadget Chain

ObjectInputStream.readObject()
HashMap.readObject()
HashMap.hash()
URL.hashCode()
URLStreamHandler.hashCode()
URLStreamHandler.getHostAddress()
URL.getHostAddress()
InetAddress.getByName() // sink

hosts corresponding IP address. The entire call stack is summarized in Table 1. Gadget chains can become much more complex and also have more serious implications (see Section 2.2) than making an application perform an arbitrary DNS request [2].

```
1  public synchronized int hashCode() {
2  /* l. 1156 - 1158 omitted */
3      hashCode = handler.hashCode(this);
4      return hashCode;
5  }
6  synchronized InetAddress getHostAddress() {
7  /* l. 987 - 994 omitted */
8      hostAddress = InetAddress.getByName(host);
9  /* l. 996 - 999 omitted */
10 }
```

Listing 3: URLDNS gadget in URL [3]

```
1  protected int hashCode(URL u) {
2  /* l. 364 - 371 omitted */
3      InetAddress addr = getHostAddress(u);
4  /* l. 373 - 397 omitted */
5  }
6  protected InetAddress getHostAddress(URL u) {
7      return u.getHostAddress();
8  }
```

Listing 4: URLDNS gadgets in URLStreamHandler [4]

2.1 The Ysoserial Exploit Repository

Ysoserial is a repository containing known gadget chains and a tool for generating payloads out of the aforementioned gadget chains [29]. In this context, a payload is the implementation of a gadget chain as a serialized Java object. The latest (at the time of writing) *Ysoserial* release version, *v0.0.6*, implements 33 gadget chains. An additional ten gadget chains exist within the *newgadgets*-branch: *Atomikos*, *Ceylon*, *CommonsCollections8*, *CommonsBeanutils2*, *Clojure2*, *CreateZeroFile*, *Jython2*, *Ssrf*, *SpringJta*, *Struts2JasperReports*. Furthermore, one can find four additional gadget chains related to *jython-standalone*, *jython*, *rome*, and *wildfly-connector* in unresolved pull requests[3]. We denote these as the payloads *Jython3*, *Jython4*, *ROME2* and *WildFly* respectively. All of these gadget chains could be verified on a simple vulnerable application, and thus they are included in the experiment. *Ysoserial* organizes its gadget chains in the *src/main/java/ysoserial/payloads/* subdirectory. For most[4] payloads the annotation *@Dependencies()* denotes the dependencies introducing the gadgets with exactly one dependency version for which the exploit is known to succeed. As shown by *Sayar et al.* [42], the affected dependency versions are more accurately depicted as a range from the point of a vulnerable gadget being introduced in a library to the patch thereof. We build on top of the previous work (19 gadget chains), by considering all 47 *Ysoserial* gadget chains and including 45 thereof in the experiment (see Section 3).

2.2 Impact Analysis of deserialization exploits

In addition to *Ysoserial*, we also include *CVE-2022-36944* as the only Proof-of-Concept we could find for a gadget chain not contained within the repository. This leads to a total of 48 gadget chains. As presented in Table 2, the majority (32) of exploits directly result in **Remote Code Execution**. Note that payloads with multiple gadget chain variations are summarized as bracketed ranges in the table. The payloads marked with an asterisk denote the gadget chains used in the experiment by *Sayar et al.* [42].

While requiring an additional step for preparation, the payload for **Python Script Execution** ultimately results in RCE as well, since it permits writing an arbitrary Python script and then executing it. **Remote Class Loading** payloads load a compiled Java class from a remote server and instantiate it. Without outbound firewall rules this also has the same effect as Remote Code Execution.

The **File Upload** payloads can be used to create a new file on the target machine. With *AspectJWeaver*, the attacker has control over both the filename and the file content. *FileUpload1* and *Wicket1* also write arbitrary data to a directory chosen by the attacker. Yet, the target file name can only be random and is consequently difficult to guess. This makes it easier to find the uploaded data externally when using *AspectJWeaver* so that, for instance, a crafted JSP document can be executed on a webserver (see implications of unrestricted File uploads from OWASP [21]).

File Modification behaves slightly different than the *File Upload payloads* in the sense that an attacker can define the file name but not the content. As pointed out in the individual payload source code of both *CreateZeroFile* and *CVE-2022-36944* this could lead to denial of service through overwriting or erasing vital files.

Java Expression Language (EL) is used for connecting a web application's presentation layer with the application logic [34]. For example, a Java Server Page (.jsp) could invoke a function inside a managed Java Bean or make use of a set of functions available by default within EL. The adverse usage of arbitrary EL expressions is summarized under CWE-917 as *Expression language Injection*, which, depending on the server technologies in use, can result in data leakage, tampering server-side logic, or even RCE [11, 36]. In particular, this becomes evident with the payload *Myfaces2*, which utilizes the same gadget chain as *Myfaces1* to load a remote class using Expression Language (see also Section 3.2).

Even with the patches to remediate loading remote class bases over RMI and LDAP in place [9], the **Java Naming and Directory Interface** (JNDI) still can be leveraged for Remote Code Execution,

[2]The DNS request seems harmless, but can be used to confirm whether an application is generally vulnerable to insecure deserialization as shown in [13].
[3]See [29] pull requests 200, 153, 167 and 177.

[4]With the exception of *Hibernate1* and *Hibernate2*, which need slightly differing dependencies to be present, depending on the version of Hibernate being used.

Table 2: Impact of individual gadget chains

Impact	Payload
RCE	*BeanShell1**, Ceylon, *Click1**, *Clojure1**, Clojure2, *CommonsBeanutils1**, CommonsBeanutils2, *CommonsCollections[1-7]**, CommonsCollections8, *Groovy1**, JBossInterceptors1, JSON1, Javassist-Weld1, *Jdk7u21**, Jython[2-4], *MozillaRhino[1-2]**, *ROME**, ROME2, *Spring[1-2]**, Struts2JasperReports, *Vaadin1**, Hibernate1
Python Script	Jython1
Remote Class	C3P0, Myfaces2
File Upload	AspectJWeaver, FileUpload1, Wicket1
File Modification	CreateZeroFile, CVE-2022-36944
EL invocation	Myfaces1
JNDI-Lookup	Atomikos, Hibernate2, SpringJta, WildFly
Blind SSRF	Ssrf
Detection	JRMPClient, JRMPListener, URLDNS

given the existence of exploitable *ObjectFactories* on the target applications classpath. As shown in [45], this can, for instance, be achieved with the *BeanFactory* class, commonly available on Apache Tomcat Servers. **Blind Server Side Request Forgery** has the target application invoke an arbitrary web request to an internal or external service.

The payloads *JRMPClient* and *JRMPListener* are also related to RMI, however, their impact is not as severe as object lookup payloads. *JRMPClient* opens up a connection to a remote registry, but without the existence of stubs on the target application, it will not be possible to have the application load an arbitrary class from an attacker-controlled proxy. As stated within the payload, it can however, be employed for DoS. *JRMPListener* opens up an arbitrary port. Finally, *URLDNS* invokes an arbitrary DNS lookup as shown in the example in Section 2. Since neither *JRMPClient*, *JRMPListener* or *URLDNS* require any dependencies on the target application classpath, verifying their success is easy, and their impact is rather low; these payloads are most likely used for the *detection* of a deserialization entry point.

3 EXPERIMENTATION

We visualize our experiment setup in Figure 1. The capital letters of the outlining boxes distinguish the main phases of (A) collecting the data for the experiment, (B) preparing and executing the individual gadget chains on a vulnerable application, and (C) processing the resulting raw data for the purpose of analysis and development of our tool. In the following sections, we elaborate in more detail on the implementation specifics. The result of running the experiment is a collection of JSON files (one per payload) containing all exploitable JDK-dependency combinations.

3.1 Preparation

External dependencies represent 44 out of 48 gadget chains (e.g., *CommonsCollections1* requires the *commons-collections* library). To determine the full version range for which a dependency enables a gadget chain, we need to download all releases of these dependencies (Figure 1, 1a). Towards this end, a total of **5,455** dependency JAR files from 53 libraries were scraped from the Maven repository. Note that occasionally newer versions of dependencies get relocated to a different directory in Maven. For example, *javax.servlet/servlet-api* relocated to *javax.servlet/ javax.servlet-api* and later to *jakarta.servlet/jakarta.servlet-api*. These dependency variations technically are distinct major releases of the same dependency and are considered as such in the experiment, meaning that if a payload includes *servlet-api* all variations are tested against it.

We download a total of 244 JDKs (1b): 122 JDK versions from the Oracle archive [6], 38 from IBM [8], 35 from the OpenJDK archive [2], and 49 from the Adoptium[5] archive [25]. Note that we leave out the Oracle JDK versions 8 and 11 due to the new OTN license [10]. However, one can estimate from the corresponding OpenJDK and Temurin-JDK versions, whether a payload would also apply to the corresponding Oracle JDK.

We use three different JAR files for payload generation: two individual builds of *Ysoserial* and the built CVE-2022-36944 PoC (Figure 1, 2a). With some minor modifications [6] the payloads from the newgadgets branch and the four payloads in open pull requests can be ported to the main version of the *Ysoserial* tool. Additionally we rebuild *Ysoserial* with *Hibernate5* to have the *Hibernate* payloads function for Hibernate versions 5 and above, as shown in [44]. The *Hibernate* payloads for earlier versions of *hibernate* will still use the regular build of *Ysoserial*.

Each payload is run for each of the 244 JDKs and combinations of dependency versions. For maximum compatibility, we use the same JDK for generating the *Ysoserial* payloads as for compiling and running the target class (3a, 3b). However, beginning with Java 16 illegal reflective access is denied, an operation heavily used in *Ysoserial*. So for JDK versions 16 and up, we craft the serialized payload with the most recent Java version (e.g. JDK-15.0.2), which permits that option [28].

To determine whether a payload exploits a given library-JDK combination, we generate the exploit code so that it creates a proof text file on success. We accommodate for parallel execution by creating a unique filename from which we can later determine the payload-, JDK-, and -dependency-version(s)-name. In regard to the RCE payloads, the file creation is done trivially by executing a shell *touch* command. For the remaining payloads, we perform additional setup steps (3c) to write a file upon detection of successful exploitation. After running the experiment, we collect all generated files and parse them into individual result files per payload (4a).

3.2 Verifying Non-RCE Payloads

As the directory location can be freely chosen in the File Upload payloads, these can be verified in a similar manner to the RCE payloads. The Python script for the *Jython1* payload directly executes a command on the CLI which then creates the file. Specifically for *CVE-2022-36944*, junk data is written into an existing file and after the successful payload execution is confirmed by checking whether

[5]The Eclipse Adoptium project provides the Temurin-JDK as an alternative open-source JDK [14].
[6]Such as moving the payloads *Ssrf* and *CreateZeroFile* to their own class definition files.

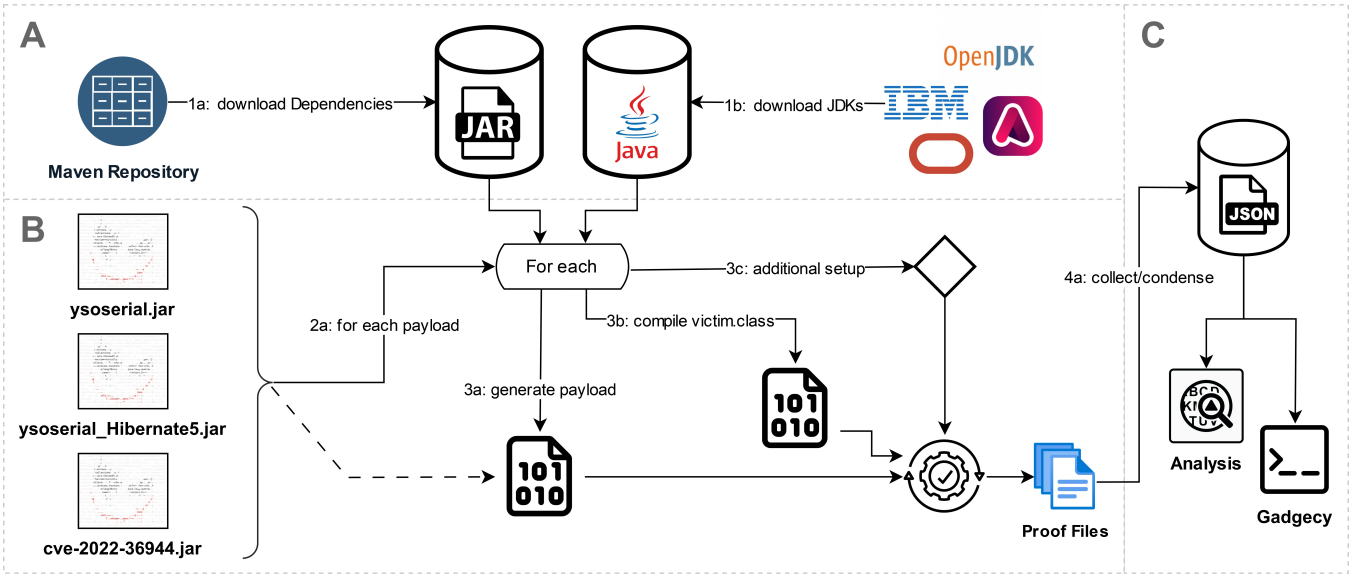


Figure 1: Experimentation framework

the file contents have been erased. The verification of all JNDI-lookup payloads, *C3P0*, *JRMPClient* and *Listener* can be abstracted through either receiving incoming socket requests triggered by the payload or connecting to a port thereby opened and on success creating the file for later retrieval. To speed up the process of testing the *Ssrf* payload we run a simple Python webserver and have the payload query a (non-existing) subdirectory named after the library and JDK version used. We then collect the results from the web server logs. For the *URLDNS* payload, we modify the vulnerable Java class to use the local machine address as a DNS server and capture incoming requests with a UDP socket server. Setting up an environment for testing *Myfaces1* requires adding a managed Bean class to a web service and deploying it on an application server. Towards this end, one would need to run a containerized application server and mount the */tmp* and *webapps* directory to the host, to verify exploitation and redeploy without having to restart the container. Note that the web application also needs to expose a serialization entry point. Using this setup we were able to confirm the *Myfaces1* payload as being functional. However, we omit both *Myfaces1* and *Struts2JasperReports* from our experiments, since the results would be prone to false positives and negatives depending on the web application server (and version) being used.

We were unable to construct a working PoC for the payload *Myfaces2*. *Myfaces2* relies on the same gadget chain as *Myfaces1* but creates a specific Expression Language statement to be executed using *Myfaces1*, which upon execution may result in instantiating a remote class. That is to say in terms of dependency and JDK requirements an application susceptible to *Myfaces1* is also susceptible to *Myfaces2* and vice-versa[7]. With the omission of *Myfaces1* and *Struts2JasperReports* we include 46 of 48[8] gadget chains in our experiment.

3.3 Running the Experiment

For a given payload with required libraries l_i, individual number of versions $ver(l_i)$ and number of JDKs $j = 244$ in the experimentation dataset a total number of

$$244 * ver(l_1) * ... * ver(l_n)$$

would have to be run in order to capture every possible library-JDK combination. Taking payloads such as *JSON1* into consideration this would mean running

$$244 * 25 * 259 * 1 * 11 * 32 * 11 * 22 * 259 * 22 \approx 7,67 * 10^{14}$$

payload executions. This is neither feasible nor necessary (see also [42]). The *Ysoserial* repository explicitly gives a combination of library versions for which the exploit is known to function. Therefore, we variate only one library version of this combination at a time and fixate the other libraries to their proven version, resulting in:

$$244 * (ver(l_1) + ... + ver(l_n))$$

executions. Thus, for the gadget chains used in the experiment, the upper bound for individual runs lies at $244 * (259 + 251 + 220 + 260 + 1 + 11) = 244,732$ when generating payloads with *SpringJta*.

The experiment is run on the Debian Linux 5.10.197 OS, with a 64-core AMD EPYC 7713P processor (2.00 GHz) and 995 GB of RAM. All code is written in Python, with the exception of the Java classes used for verifying the payloads. We leverage GNU Parallel for multi-threaded execution.

[7]Although *Myfaces1* provides more flexibility towards the EL expression being used and thus in general will more often be successful with real applications.

[8]As explained *Myfaces2* cannot be described as a unique gadget chain, reducing the total amount of gadget chains from 49 to 48.

4 EVALUATION

We systematically evaluate the experiment results in accordance with our research questions. In RQ1, we discuss immediate findings and how to interpret the experimentation data. This shows the specific combinations of JDKs and dependencies required for an exploitable gadget chain to be present. For 71,74% of gadget chains, we are also able to condense and transform the results into an easy-to-read, tabular format. Thereafter, we use the previous results to highlight the difficulties of gadget chain disclosure in RQ2 and the frequency of gadget chains being present in Java projects in RQ3. Our aim is to verify our expectation that the general awareness of gadget chains is low, whilst simultaneously showing that their occurrence is widespread.

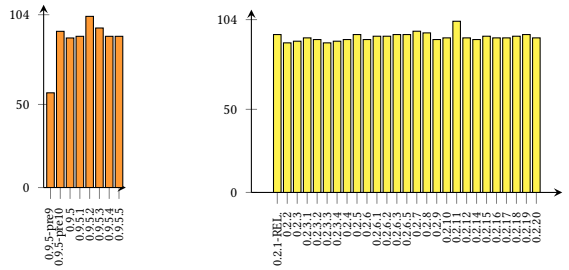
> **RQ1:** What are the dependency and JDK prerequisites for known gadget chains?

The raw experimentation data is composed of entries forming a strict relationship between specific dependency and JDK versions. We suspect that if one JDK is vulnerable to a payload, the dependency prerequisites for all other applicable JDKs will be the same. In doing so we decouple JDK and dependency requirements. Given L_i as the set of library versions vulnerable to a specific payload when run on JDK i. We can calculate the subset of all library versions common amongst JDKs and compare it to the superset of all distinct library versions captured across all JDKs.

$$L_{common} = \bigcap_{i \in JDK} L_i \quad \text{and} \quad L_{all} = \bigcup_{i \in JDK} L_i$$

For 33 (71,74 % of) payloads L_{common} and L_{all} are equal. Table 3 summarizes the applicability of these exploits on JDK and library versions in a condensed format. The asterisk (*) denotes that all versions are affected. For ranges such as *aspectjweaver[1.7.2 - *]*, this is to be interpreted as all versions beginning from 1.7.2 are affected. To give an example, Table 3 can be used to lookup the prerequisites for the gadget chain *CommonsBeanutils2*. Given a Java application exposing an insecure deserialization entry point, the payload will lead to exploitation if the application satisfies the following prerequisites. The dependency *commons-beanutils* in version 1.9.0 or later and any version of *commons-logging* is present on the application's classpath, and it is running on an applicable JVM. As denoted in the JDK column, this could be either: any update of the Oracle- or OpenJDK up to version 15, or the IBM- and Temurin-JDK in versions 8 and 11.

For the remaining 13 (28,26%) examined payloads, there is a variation in the dependency versions used for exploitation among susceptible JDKs. Hence, Table 3 cannot properly illustrate all possible combinations. For example, the *C3P0* gadget chain was found to work on 104 JDKs, but only 57 of these JDKs could be exploited with the 0.9.5-pre9 version of the *c3p0* dependency. Figures 2a and 2b show the discrepancies for both dependencies involved in the *C3P0* gadget chain. Variations of this kind may occur either due to network errors when testing non-RCE payloads or incompatibilities of newer dependency versions with old JDK versions. For instance, we observe the latter for *Vaadin1*, where no exploitation is possible



(a) c3p0 **(b) mchange-commons**

Figure 2: Occurence of individual library versions in successful C3P0 payload exploitation (104 JDKs)

using *vaadin-shared* starting from version 8.0.0 when testing on Oracle JDK 7. When comparing the manifest files of *vaadin-shared* before and after version 8, one finds that beginning with version 8.0.0 the manifest defines a *Build-Jdk* of 1.8.0_60 or higher. Consequently, this results in an *unsupported major.minor version* error during execution on updates of Oracle JDK 7. For this reason, the application crashes before it is able to execute the gadget chain and thus no exploitation can be observed. We refer to our repository in Section 8, containing the experiment results for all 46 payloads, including those not displayed in Table 3.

Analyzing the results unveils three immediate findings. For one, gadget chain attacks are applicable to all JDK versions. Specifically, the *CommonsCollections6* and *-7* payloads successfully exploit all 244 JDKs used in the experiment. Next, it also shows that an attacker can use a combination of the payloads used for detecting deserialization entry points to cover the full range of JDKs (without any dependency needing to be present). As a third finding, we observe that the payloads *AspectJWeaver*, *Ceylon*, *CommonsBeanutils2*, *C3P0*, *Jython3*, *Jython4 Vaadin1*, *WildFly* and *SpringJta*[9] rely solely on most-current dependency versions.

> **RQ2:** How are gadget chains being disclosed?

We find that the CVE structure hardly is an ideal solution for reporting dependencies enabling gadget chains for two reasons:

(1) Gadget chains are not vulnerabilities as is. They may only be leveraged in combination with an insecure deserialization entry point toward exploitation.
(2) When gadget chains are composed of gadgets originating from multiple different dependencies, it becomes difficult to capture these within a CPE identifier, which only accommodates the reporting of one specific configuration in isolation.

Nevertheless, the Maven repository lists CVEs for dependencies such as *commons-collections*, clearly denoting insecure deserialization [7, 39]. Thus, to prove our point, we scrape the CVE identifiers of all library versions involved in executing a *Ysoserial* payload

[9]Vaadin-server 7.7.42 and Spring-core 5.3.31 are in this case recent versions since vaadin 7.7.x and spring 5.3.x are still receiving releases, independently from the newer major releases [7].

Table 3: Payloads with no library variation among applicable JDK versions

Payload	Libraries	JDK
AspectJWeaver	aspectjweaver[1.7.2 - *], commons-collections[20040616, 3.1 - 3.2.2]	oracle-jdk-[9-*]-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
BeanShell1	bsh[2.0b5]	oracle-jdk-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
Ceylon	ceylon.language[*]	jdk-[7-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
Clojure1	clojure[1.8.0, 1.8.0-RC5]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
Clojure2	clojure[1.8.0, 1.8.0-RC5]	oracle-jdk-[*], temurin-jdk-[*], openjdk-[*]
CommonsBeanutils1	commons-beanutils[1.9.0 - *], commons-collections[* - 3.2.2], commons-logging[*]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
CommonsBeanutils2	commons-beanutils[1.9.0 - *], commons-logging[*]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
CommonsCollections1	commons-collections[20040616, 3.1 - 3.2.1]	oracle-jdk-[6,7]-[*]
CommonsCollections2	commons-collections4-[4.0]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
CommonsCollections3	commons-collections[20040616, 3.1 - 3.2.1]	oracle-jdk-[6,7]-[*]
CommonsCollections4	commons-collections4[4.0]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
CommonsCollections5	commons-collections[20040616, 3.1 - 3.2.1]	oracle-jdk-[7-14]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-14]-[*]
CommonsCollections6	commons-collections[20040616, 3.1 - 3.2.1]	oracle-jdk-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
CommonsCollections7	commons-collections[20040616, 3.1 - 3.2.1]	oracle-jdk-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
CommonsCollections8	commons-collections4[4.0]	oracle-jdk-[7-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[11,8]-[*], openjdk-[9-15]-[*]
CreateZeroFile	scala-library[2.12.3 - 2.12.7]	oracle-jdk-[9 - *], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
Groovy1	groovy[2.3.0-beta-2 - 2.4.0-beta-4]	oracle-jdk-[6-13], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-13]-[*]
Jdk7u21	-	oracle-jdk-6-[*], oracle-jdk-7-[*-21]
JRMPClient	-	oracle-jdk-[*], temurin-jdk-[*], openjdk-[*]
JRMPListener	-	oracle-jdk-[6 - 16]-[*], ibm-jdk-[8,11,16]-[*], temurin-jdk-[8,11,16]-[*], openjdk-[9 - 16]-[*]
Jython1	jython-standalone[2.5.2 - 2.5.4-rc1]	oracle-jdk-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[*], openjdk-[*]
Jython2	jython-standalone[2.5.2 - 2.5.4-rc1]	oracle-jdk-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[*], openjdk-[*]
Jython3	jython-standalone[2.7.3b1 - *]	oracle-jdk-[12-*]-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[11-*]-[*]
Jython4	jython[2.7.3b1 - *]	oracle-jdk-[12-*]-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[11-*]-[*]
MozillaRhino1	js[1.7R2]	oracle-jdk-[7u45-14]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-14]-[*]
MozillaRhino2	js[1.7R2]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
ROME	rome[0.5 - 1.0]	oracle-jdk-[6-15]-[*], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-15]-[*]
ROME2	rome[0.5 - 1.0]	oracle-jdk-[7u45 - 14.0.2], ibm-jdk-[8,11]-[*], temurin-jdk-[8,11]-[*], openjdk-[9-14]-[*]
Spring1	spring-core[4.0.1 - 4.2.2].RELEASE, spring-beans[3.0.0 - 4.3.30].RELEASE	oracle-jdk-[6,7]-[*]
Spring2	spring-core[4.0.1 - 4.2.2].RELEASE, spring-aop[3.0.0 - 4.2.9].RELASE, aopalliance[*], commons-logging[*]	oracle-jdk-[6,7]-[*]
Ssrf	scala-library[2.12.3 - 2.12.7]	oracle-jdk-[9 - *], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
URLDNS	-	oracle-jdk-[7 - *]-[*], ibm-jdk-[*], temurin-jdk-[*], openjdk-[*]
CVE-2022-36944	scala-library[2.13.0-M5-6e0cba7 - 2.13.8]	oracle-jdk-[12 - *], ibm-jdk-[11 - *], temurin-jdk-[11 - *], openjdk-[11 - *]

from the Maven repository website [7]. Then we query the NVD API using the collected CVEs. We consider a gadget chain not to be disclosed using the CVE if it is possible to find a combination of dependency versions, for which none of the involved dependencies has an associated CVE. Thereby, we determine whether a CVE is related to a gadget chain by querying the CVE description for the strings *seriali(s|z)e*, *seriali(s|z)ation*[10] or *marshall* or if it contains the corresponding CWE-502 weakness identifier. The 14 gadget chains for which this yields true are listed in category (1) of Table 4, with the associated CVEs listed in Appendix A. The dependencies of another five gadget chains contain dependency versions covered by CVEs, which are however generally unrelated to insecure deserialization (category (2)). In contrast, 27 (58,70%) gadget chains are uncovered by CVEs (3). This shows that developers cannot rely on CVEs to highlight weaknesses in the dependencies used for a Java project. With our work we attempt to bridge this gap by developing a tool to detect the existence of dependencies leading to known gadget chains being included in a Java application.

Table 4: CVE coverage of *Ysoserial* payloads

(1) direct: BeanShell1, CommonsCollections[1-8], CVE-2022-36944, Groovy1, Jdk7u21, Jython[1-2]
(2) indirect: Hibernate[1-2], Spring[1-2], FileUpload1
(3) uncovered: AspectJWeaver, Atomikos, C3P0, Ceylon, Click1, Clojure[1-2], CommonsBeanutils[1-2], CreateZeroFile, Jython[3-4], MozillaRhino[1-2], ROME[1-2], URLDNS, JRMPClient, JRMPListener, JavassistWeld1, JBossInterceptors1, JSON1, SpringJta, Ssrf, Vaadin1, Wicket1, WildFly

[10]Note that by searching for strings the keywords *deserialize* and *deserialization* (British and American English spelling) are also included.

RQ3: How to determine whether a given Java application contains the dependency prerequisites for insecure deserialization?

Only four (*8,67%*) gadget chains (*JRMPClient*, *JRMPListener*, *URLDNS*, *Jdk7u21*) rely solely on gadgets within the Java Class library, meaning they require no additional dependencies (see Table 3). The impact of the first three is less severe (see Section 2.2) and JDK 7u25 patches the latter [27]. Conversely, the results show that the majority (38/42) of payloads relying on additional dependencies are applicable to recent updates in LTS JDK versions. Therefore, analyzing application dependencies can be considered a reasonable approach for determining whether an application includes known gadget chains. Going from this observation, we develop *Gadgecy*, a tool for detecting these gadget-chain-enabling dependencies. *Gadgecy* operates in two modes:

- Reading directories and comparing contained JAR file hashes with the hashes of gadget-chain-enabling dependencies.
- Parsing of *pom.xml* dependency files and comparison with experiment results.

Using *Gadgecy*, we perform two follow-up experiments to inspect whether real-world projects include vulnerable dependency versions. First, we download **2,211** projects from the Apache Distribution directory [1], excluding archived or JAR-less projects. The resulting dataset includes **115,053** JAR files, meaning each project is on average composed of 52 JAR files. Furthermore, we assume that any JAR file within a project directory could be a runtime dependency included within the classpath of the core application. Table 5 summarizes the results generated by *Gadgecy*. In total, **238** (*10,76%*) projects contain the dependencies required for a gadget chain attack, including widely used artifacts like *Struts*, *Kafka*, *Hadoop*, *Druid* and *Solr* in their most recent versions. The most commonly

Table 5: Projects in Apache Distribution Directory containing prerequisites for known gadget chains

Payload	# Projects	Examples
CommonsBeanutils2	171	Druid, Hadoop, Kafka, Struts
CommonsBeanutils1	169	Druid, Hadoop, Kafka, Struts
CommonsCollections → (1,3,5,6,7)	38	Airavata, SystemDS, Pig
AspectJWeaver	31	DolphinScheduler, StreamPark
C3P0	24	Solr, Zeppelin
Lazylist	9	Drill, James server
SpringJta	8	DolphinScheduler, Ignite
FileUpload1	6	ManifoldCF, Tomahawk
CreateZeroFile & **Ssrf**	5	HugeGraph, Inlong
ROME (1,2)	4	ManifoldCF, Archiva
Jython3	4	Hop Client
MozillaRhino (1,2)	3	Pig, SXF
CommonsCollections → (2,4,8)	3	Pig, Zeppelin
Spring (1,2)	1	Zeppelin

applicable payloads are *CommonsBeanutils1* and *CommonsBeanutils2*, the latter appearing slightly more frequently since it does not require gadgets found in the *commons-collections* library. Note that payloads with the same dependency prerequisites are grouped in Table 5, as they consequently were found applicable to the same set of projects.

For the second part of the analysis, we download the *pom.xml* file structure from popular and active Java repositories. This implies downloading the pom.xml at the root of the repository and all pom.xml files defined as modules in subdirectories recursively. Doing so provides the benefit of being able to use the *Maven* CLI tool without having to clone the entire repository. We define *popular* as having a star count of 100 or above, and active as having received an update in the year 2023 while not being flagged as *archived* [31]. In addition, these repositories originate from a limited set of **95** renowned Github projects, to reduce the amount of false positive findings from irrelevant, individual Java projects (see appendix B). The resulting dataset consists of **400** repositories with a total of **13,745** pom.xml files. Notice that pom.xml files accommodate for the definition of modules in subdirectories of the repository. This explains the large discrepancy between the amount of repositories and dependency files. Given the dependency file structure of the repositories, we generate the dependency trees for all modules using the *Maven Dependency Plugin* [23] and analyze these individually. In contrast to the previous experiment, using the *Dependency Plugin* we can now accurately resolve transitive dependencies, leading to fewer false negatives. It is important to mention that according to Mavens dependency scope mechanism, all transitive dependencies, but those defined for testing, will be available during runtime [24]. Consequently, we find that **1,254** modules (*9,12%*) contain the dependency-version combinations required for the analyzed, known gadget chains. Furthermore, these 1,254 modules can be traced back to 147 (*36,75%*) of the repositories in the dataset. Again, the *CommonsBeanutils* gadget chains contribute to the largest amount of findings, with *CommonsBeanutils2* being present in 1,112 (88,68%) and *CommonsBeanutils1* in 1,104 (88,01%) of the detected modules.

We further investigate which dependencies in their respective versions are responsible for the gadget chains to be present in these Java projects. Table 6 summarizes the Top 10 (of 20 in total) dependencies found to be part of a known gadget chain within a module. For example, the *commons-collections* library is responsible for 1,180 gadget chain findings, 1,088 in version 3.2.2 and 92 in version 3.2.1 of the library. The information displayed in Table 6 is vital for the developers of these libraries since it reveals just how frequently their library is the reason for weaknesses in real-world applications. Towards application developers, it shows which dependency patches to prioritize, should there be a newer version not containing the gadgets used in a known gadget chain.

Leveraging *Gadgecy*, we demonstrate just how frequent (*10,76%* and *9,12%*) gadget chains appear in popular Java projects. Note that these Java projects are often dependencies themselves or frameworks, meaning they are likely used by other Java applications, thus passing on gadget chains downstream. From a developer's perspective, this implies the importance of knowing which gadget chains can potentially be used against their applications and hardening deserialization entry points accordingly (see Section 5.2). In general, a fatal discrepancy between the potential impacts of insecure deserialization (see Section 2.2), insufficient reporting mechanisms and widespread gadget-dependency usage becomes evident. With the tool *Gadgecy* (and the underlying dataset) it is possible to combat the lack of visibility towards this issue.

5 RELATED WORK

Frohoff and *Lawrence* first highlighted the inherent danger of insecure object deserialization in a talk in 2015 [30]. It was later shown that Java deserialization flaws are not limited to native Java object serialization, but also other serialization libraries such as *Kryo*, *XStream* or *SnakeYAML* [15]. *Sayar et al.* analyzed how gadgets are introduced into Java libraries and how vendors patch deserialization vulnerabilities [42].

5.1 Tools for detecting Deserialization Vulnerabilities

Tools for detecting new gadget chains have been developed in [17–19, 32, 40]. Other tools aid in identifying the existence of a deserialization entry point within a given application. *Java deserialization Scanner* [22] is a Burp Suite add-on that implements 20 hard-coded payloads from the *Ysoserial* repository. *Koutroumpouchos et al.* [35] developed a command line tool for detecting web-based entry points for deserialization. As such they focus only on detecting whether such an entry point exists and not if there actually is an applicable payload for exploitation. *Joogle* is a tool which helps in finding interesting attack gadgets by searching for classes overriding magic methods [20]. Insecure deserialization is not limited to Java. Similar projects to *Ysoserial* exist for .NET [37] and PHP [12]. Similar to *Ysoserial*, these repositories collect known gadget chains and provide a tool for automatic payload generation.

Table 6: Top 10 dependencies enabling gadget chains in Github project dataset

Dependency	# Modules	# Repositories	Versions
commons-collections	1180	137	('3.2.2', 1088), ('3.2.1', 92)
commons-logging	1118	127	('1.2', 698), ('1.1.1', 211), ('1.1.3', 204), ('1.0.3', 3), ('1.0.4', 2)
commons-beanutils	1112	123	('1.9.4', 960), ('1.9.2', 127), ('1.9.3', 25)
aspectjweaver	360	22	('1.8.13', 127), ('1.9.7', 119), ('1.9.6', 65), ('1.8.9', 15), ('1.8.10', 8), ('1.9.21', 8), ('1.9.20.1', 5), ('1.9.0', 4), ('1.9.20', 3), ('1.8.6', 2), ('1.9.5', 2), ('1.9.9.1', 2)
servlet-api	72	10	('2.5', 32), ('3.1.0', 21), ('4.0.1', 15), ('2.4', 4)
spring-context	58	14	('5.3.22', 19), ('5.2.23.RELEASE', 16), ('5.3.30', 4), ('5.3.27', 4), ('4.3.9.RELEASE', 4), ('5.2.21.RELEASE', 3), ('5.3.31', 2), ('4.3.20.RELEASE', 1), ('4.3.30.RELEASE', 1), ('4.3.7.RELEASE', 1), ('5.3.20', 1), ('5.1.1.RELEASE', 1), ('5.3.2', 1)
spring-beans	57	14	('5.3.22', 19), ('5.2.23.RELEASE', 16), ('5.3.30', 4), ('4.3.9.RELEASE', 4), ('5.3.27', 3), ('5.2.21.RELEASE', 3), ('5.3.31', 1), ('4.3.20.RELEASE', 1), ('4.3.30.RELEASE', 1), ('4.3.7.RELEASE', 1), ('5.3.20', 1), ('5.1.1.RELEASE', 1), ('5.3.18', 1), ('5.3.2', 1)
spring-core	56	14	('5.3.22', 18), ('5.2.23.RELEASE', 16), ('5.3.30', 4), ('5.3.27', 4), ('4.3.9.RELEASE', 4), ('5.2.21.RELEASE', 3), ('5.3.31', 1), ('4.3.20.RELEASE', 1), ('4.3.30.RELEASE', 1), ('4.3.7.RELEASE', 1), ('5.3.20', 1), ('5.1.1.RELEASE', 1), ('5.3.2', 1)
spring-tx	53	14	('5.3.22', 16), ('5.2.23.RELEASE', 14), ('5.3.30', 4), ('5.3.27', 4), ('4.3.9.RELEASE', 4), ('5.2.21.RELEASE', 3), ('5.3.31', 1), ('4.3.20.RELEASE', 1), ('4.3.30.RELEASE', 1), ('4.3.10.RELEASE', 1), ('5.3.20', 1), ('5.1.1.RELEASE', 1), ('5.0.6.RELEASE', 1), ('5.3.2', 1)
spring-context-support	41	12	('5.2.23.RELEASE', 16), ('5.3.22', 6), ('5.3.30', 4), ('4.3.9.RELEASE', 4), ('5.3.27', 3), ('5.2.21.RELEASE', 3), ('5.3.31', 1), ('4.3.30.RELEASE', 1), ('4.3.10.RELEASE', 1), ('5.3.20', 1), ('5.3.2', 1)

5.2 Prevention

It was analyzed in [42] how vendors remediate deserialization vulnerabilities in their applications. The most frequent solutions are to add an allow or deny list in alignment with JDK Enhancement Proposal (JEP) 290 *ObjectInputFilter*s [41], remove the *Serializable* interface from gadgets, harden or remove the deserialization entry point or update the libraries associated with the gadget chain used for exploitation. With this work, we show that the latter option will not be viable for some gadget chains. Backward-compatibility breaking changes, beginning with JDK version 16, made accessing internal parts of the JDK, using reflection through unnamed modules, denied per default [33]. Amongst others, the *Commons-Beanutils* gadget chains, which were found to be most common amongst Java projects in our experiment, rely on this mechanism and can thus be remediated by migrating to JDK version 16 or newer. However, at best, only trusted data should be serialized. All other remediation strategies come with individual drawbacks, such as maintaining *ObjectInputFilters*, so that they include all applicable gadget chains without breaking the application logic.

6 LIMITATIONS

The scope of this publication is limited to the gadget chains known in the *Ysoserial* repository [29] and the PoC for CVE-2022-36944 [46]. We are not aware of any other currently available resources for publicly known gadget chains. Furthermore, Oracle JDK versions 8 and 11 are excluded from the study due to the OTN license. Going from the results of the other JDK vendors the impact can be estimated, but not definitely confirmed for these Oracle JDK versions. Also, note that the study is confined to the native Java Object Serialization. Other serialization mechanisms and formats exist (e.g. XML, JSON, YAML), however, these come with their own sets of vulnerabilities, as was analyzed in [15].

The developed tool *Gadgecy* only checks whether the dependencies introducing the gadgets used for a *Ysoserial* payload are present. For the results to translate to a security implication an execution path to *ObjectInputStream.readObject()* using untrusted input is required to be present. Tools and frameworks to achieve this task already exist [16, 43]. However, should the insecure deserialization point originate from an external library or framework, this type of analysis leads to false negatives. Finally, it needs to be mentioned that our results are a snapshot of currently available gadget chains.

7 CONCLUSION

When dealing with deserialization vulnerabilities, it needs to be acknowledged, that they derive from an unfortunate constellation of individually harmless software properties. Without the presence of either gadget-introducing dependencies or a reachable deserialization entry point, no vulnerability exists to begin with. However, in combination, the consequences are severe, more often than not leading to remote code execution. This work demonstrates that gadget-chain-enabling dependencies are in fact widespread, as they appear in 10,76% and 9,12% of well-known Java projects within the respective datasets. In particular, we find that popular Java dependencies like *Commons Beanutils*, *C3P0* and *AspectJWeaver* include the gadgets required for an attack in their most recent versions.

Furthermore, our research reveals the majority (58,70%) of known gadget chains as being entirely uncovered by CVEs. Seeing that the CVE is not an ideal reporting mechanism for gadget chains raises the question of how to make developers aware of the underlying issue. Towards this end, we propose the tool *Gadgecy* for determining whether a given Java project contains the dependency requirements for a deserialization vulnerability. Its lightweight analysis and simplicity enable easy integration in a CI/CD pipeline. *Gadgecy* operates on built projects through checksum comparison of JAR files contained within or on POM build files. The tool is an abstraction layer on top of the data collected by our experiments, providing a nearly complete picture of dependency (-version) prerequisites for known gadget chain attacks.

Our work aims to increase awareness towards Java deserialization vulnerabilities by showing that the dependency prerequisites are easily fulfilled for a gadget chain attack. This renders the Java standard serialization protocol unsafe without any further mitigation techniques in place. We provide a dataset mapping 46 known gadget chains to the JDK and dependency versions they are applicable to. As such our dataset more than doubles the amount of inspected gadget chains (46) compared to state-of-the-art (19). We direct our further research at developing tools for the automatic discovery of new gadget chains. The inspected, known gadget chains provide a basis for our validation dataset.

8 DATA AVAILABILITY

We share the source code used for the experiments, the datasets, results and the tool *Gadgecy* including a usage explanation with the link **https://github.com/software-engineering-and-security/Gadgecy**.

ACKNOWLEDGMENTS

We would like to thank Sabine Houy and Timothée Riom for their feedback on this paper. This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

REFERENCES

[1] [n. d.]. Apache Distribution Directory. Retrieved 2023-11-10 from https://dlcdn.apache.org/
[2] [n. d.]. Archived OpenJDK GA Releases. Retrieved 2023-11-01 from https://jdk.java.net/archive/
[3] [n. d.]. java.net.URL openjdk. Retrieved 2023-11-03 from https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/net/URL.java
[4] [n. d.]. java.net.URLStreamHandler openjdk. Retrieved 2023-11-03 from https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/net/URLStreamHandler.java
[5] [n. d.]. java.util.HashMap openjdk. Retrieved 2023-11-03 from https://github.com/openjdk/jdk/blob/master/src/java.base/share/classes/java/util/HashMap.java
[6] [n. d.]. Looking for an Older Java Release? Retrieved 2023-11-01 from https://www.oracle.com/java/technologies/downloads/archive/
[7] [n. d.]. Maven Repository. Retrieved 2023-11-01 from https://mvnrepository.com/
[8] [n. d.]. Semeru Runtime Downloads. Retrieved 2023-11-01 from https://developer.ibm.com/semeru-runtime-downloads
[9] 2018. CVE-2018-3149- Red Hat Customer Portal. https://access.redhat.com/security/cve/cve-2018-3149
[10] 2019. Oracle Java SE License. Retrieved 2023-11-01 from https://www.oracle.com/downloads/licenses/javase-license1.html
[11] 2022. Expression Language Injection | OWASP Foundation. https://owasp.org/www-community/vulnerabilities/Expression_Language_Injection
[12] 2023. PHPGGC: PHP Generic Gadget Chains. https://github.com/ambionics/phpggc original-date: 2017-07-03T08:54:25Z.
[13] Philippe Arteau. 2017. Detecting deserialization bugs with DNS exfiltration. https://www.gosecure.net/blog/2017/03/22/detecting-deserialization-bugs-with-dns-exfiltration/
[14] Wayne Beaton and Eclipse Foundation. 2020. Eclipse Adoptium. https://projects.eclipse.org/projects/adoptium
[15] Moritz Bechler. 2017. Java Unmarshaller Security. (May 2017). https://raw.githubusercontent.com/mbechler/marshalsec/master/marshalsec.pdf
[16] Eric Bruneton, Remi Forax, and Eugene Kuleshov. [n. d.]. ASM. https://asm.ow2.io/index.html
[17] Sicong Cao, Biao He, Xiaobing Sun, Yu Ouyang, Chao Zhang, Xiaoxue Wu, Ting Su, Lili Bo, Bin Li, Chuanlei Ma, Jiajia Li, and Tao Wei. 2023. ODDFUZZ: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. https://doi.org/10.48550/arXiv.2304.04233 arXiv:2304.04233 [cs].
[18] Sicong Cao, Xiaobing Sun, Xiaoxue Wu, Lili Bo, Bin Li, Rongxin Wu, Wei Liu, Biao He, Yu Ouyang, and Jiajia Li. 2023. Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. https://doi.org/10.48550/arXiv.2303.07593 arXiv:2303.07593 [cs].
[19] Xingchen Chen, Baizhu Wang, Ze Jin, Yun Feng, Xianglong Li, Xincheng Feng, and Qixu Liu. 2023. Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 179–192. https://doi.org/10.1109/DSN58367.2023.00028 ISSN: 2158-3927.
[20] Narshan Dabirsiaghi. 2016. joogle. https://github.com/Contrast-Security-OSS/joogle original-date: 2015-12-15T16:46:53Z.
[21] Soroush Dalili, Dirk Wetter, Landon Mayo, and OWASP. [n. d.]. Unrestricted File Upload | OWASP Foundation. https://owasp.org/www-community/vulnerabilities/Unrestricted_File_Upload
[22] federicodotta, Jeremy Goldstein, and András Veres-Szentikrályi. 2021. Java Deserialization Scanner. https://github.com/federicodotta/Java-Deserialization-Scanner original-date: 2015-12-08T14:31:15Z.
[23] Apache Foundation. 2023. Apache Maven Dependency Plugin – Introduction. https://maven.apache.org/plugins/maven-dependency-plugin/index.html
[24] Apache Foundation. 2023. Maven – Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html
[25] Eclipse Foundation. [n. d.]. Archive Adoptium. Retrieved 2023-11-01 from https://adoptium.net/temurin/archive/
[26] OWASP Foundation. 2021. OWASP Top Ten. Retrieved 2023-10-23 from https://owasp.org/www-project-top-ten/
[27] Chris Frohoff. 2016. Java 7u21 Security Advisory. https://gist.github.com/frohoff/24af7913611f8406eaf3
[28] Chris Frohoff. 2022. Error while generating or serializing payload · Issue #176 · frohoff/ysoserial. https://github.com/frohoff/ysoserial/issues/176
[29] Chris Frohoff. 2023. *ysoserial*. https://github.com/frohoff/ysoserial original-date: 2015-01-28T07:13:55Z.
[30] Chris Frohoff and Gabriel Lawrence. 2015. AppSecCali 2015: Marshalling Pickles by frohoff. https://frohoff.github.io/appseccali-marshalling-pickles/
[31] Github. 2022. Use the REST API to manage repositories on GitHub. https://ghdocs-prod.azurewebsites.net/_next/data/j2ZDr5NDLCOPOMZWq0N1b/en/free-pro-team@latest/rest/repos/repos.json?apiVersion=2022-11-28&versionId=free-pro-team%40latest&category=repos&subcategory=repos
[32] Ian Haken. 2018. Automated Discovery of Deserialization Gadget Chains. https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf
[33] Florian Hauser. 2023. CODE WHITE | Blog: Java Exploitation Restrictions in Modern JDK Times. https://codewhitesec.blogspot.com/2023/04/java-exploitation-restrictions-in.html
[34] Eric Jendrock, Ricardo Cervera-Navarro, Ian Evans, Kim Haase, and William Markito. 2014. Java Platform, Enterprise Edition The Java EE Tutorial, Release 7. Oracle, 151–161. https://docs.oracle.com/javaee/7/tutorial/jsf-el.htm
[35] Nikolaos Koutroumpouchos, Georgios Lavdanis, Eleni Veroni, Christoforos Ntantogian, and Christos Xenakis. 2019. ObjectMap: detecting insecure object deserialization. In *Proceedings of the 23rd Pan-Hellenic Conference on Informatics (PCI '19)*. Association for Computing Machinery, New York, NY, USA, 67–72. https://doi.org/10.1145/3368640.3368680
[36] MITRE. 2013. CWE - CWE-917: Improper Neutralization of Special Elements used in an Expression Language Statement ('Expression Language Injection') (4.13). https://cwe.mitre.org/data/definitions/917.html
[37] Alvaro Muñoz. 2023. pwntester/ysoserial.net. https://github.com/pwntester/ysoserial.net original-date: 2017-09-18T17:48:08Z.
[38] NIST. [n. d.]. NVD - Home. Retrieved 2023-10-24 from https://nvd.nist.gov/
[39] NIST. 2023. NVD - CVE-2015-6420. https://nvd.nist.gov/vuln/detail/CVE-2015-6420
[40] Shawn Rasheed and Jens Dietrich. 2021. A hybrid analysis to detect Java serialisation vulnerabilities. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)*. Association for Computing Machinery, New York, NY, USA, 1209–1213. https://doi.org/10.1145/3324884.3418931
[41] Roger Riggs. 2016. JEP 290: Filter Incoming Serialization Data. https://openjdk.org/jeps/290
[42] Imen Sayar, Alexandre Bartel, Eric Bodden, and Yves Le Traon. 2023. An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities. *ACM Transactions on Software Engineering and Methodology* 32, 1 (Feb. 2023), 25:1–25:45. https://doi.org/10.1145/3554732
[43] Uwe Schindler, Jochen Schalanda, David Weiss, and Dominik Stadler. 2023. Policeman's Forbidden API Checker. https://github.com/policeman-tools/forbidden-apis original-date: 2015-03-13T22:50:03Z.
[44] Francesco Soncina (phra). 2018. Java Deserialization — From Discovery to Reverse Shell on Limited Environments. https://medium.com/abn-amro-red-team/java-deserialization-from-discovery-to-reverse-shell-on-limited-environments-2e7b4e14fbef
[45] Michael Stepankin. 2019. Exploiting JNDI Injections in Java. https://www.veracode.com/blog/research/exploiting-jndi-injections-java
[46] yarocher. 2023. lazylist-cve-poc: POC for the CVE-2022-36944 vulnerability exploit. https://github.com/yarocher/lazylist-cve-poc

A DESERIALIZATION CVES

- **bsh**: CVE-2016-2510
- **commons-collections**: CVE-2015-7501, CVE-2015-6420
- **commons-beanutils**: CVE-2019-10086
- **commons-fileupload**: CVE-2013-2186
- **groovy**: CVE-2016-6814, CVE-2015-3253
- **spring-core**: CVE-2011-2894
- **jython-standalone**: CVE-2016-4000
- **scala-library**: CVE-2022-36944

B GITHUB PROJECTS

Activiti, airbnb, alibaba, Alluxio, android, androidannotations, AntennaPod, antlr, apache, apolloconfig, arduino, auth0, Baseflow, bazelbuild, beemdevelopment, Blankj, btraceio, bumptech, clojure, commons-app, dbeaver, deeplearning4j, dromara, eclipse, elastic, facebook, flyway, geogebra, google, GoogleContainerTools, graphql-java, greenrobot, hazelcast, hibernate, JabRef, java-decompiler, jenkinsci, jitsi, journeyapps, junit-team, jwtk, keycloak, languagetool-org, lettuce-io, libgdx, lingochamp, material-components, MinecraftForge, mockito, mybatis, NanoHttpd, neo4j, Netflix, netty, nostra13, OpenRefine, openzipkin, oracle, PhilJay, pinpoint-apm, processing, projectlombok, provectus, quarkusio, questdb, ReactiveX, realm, redis, redisson, resilience4j, roboelectric, seata, SeleniumHQ, signalapp, sirixdb, skylot, SonarSource, sonatype, spring-projects, square, StarRocks, strongbox, supertokens, TEAMMATES, TeamNewPipe, Tencent, termux, TheAlgorithms, thingsboard, trello, xwiki, Yalantis, zaproxy, zendesk, zxing

See section 8 for repository names and commit-hashes