Counterfeit Object-Oriented Programming Vulnerabilities: An Empirical Study in Java

Joanna C. S. Santos
joannacss@nd.edu
University of Notre Dame
Notre Dame, IN, USA

Xueling Zhang
xueling.zhang@rit.edu
Rochester Institute of Technology
Rochester, NY, USA

Mehdi Mirakhorli
mxmvse@rit.edu
Rochester Institute of Technology
Rochester, NY, USA

ABSTRACT

Many modern applications rely on Object-Oriented (OO) design principles, where the basic system components are objects and classes. They share objects with other processes, store them in disk/files for future retrieval or transport them over network to other systems. Object-oriented programs leverage numerous dynamic features and design principles such as runtime dispatching and object-oriented callbacks which allow flexible software design. Although seemingly innocuous, these features can be abused by the attackers to hijack the program's control flow to an undesirable behavior. This is referred to as Counterfeit Object-Oriented Programming (COOP), in which attackers hijack objects in the program in order to create a sequence of method calls that introduce a malicious behavior. COOP is a type of code reuse attack in which a hacker hijacks objects (gadgets) in the program and use that to control the program execution flow via manipulating the sequence of methods and data being passed among these methods (gadget chains). In this paper, we describe a preliminary empirical investigation of COOP attacks in real software systems caused by untrusted object deserialization. In this preliminary study, we investigated the severity of these attacks, their consequences, and how they were mitigated by developers. Furthermore, we used the findings to create a dataset of vulnerable software projects and their fixes.

CCS CONCEPTS

• **Security and privacy** → **Software security engineering**; • **Software and its engineering** → *Software development techniques.*

KEYWORDS

common weakness enumeration, counterfeit object-oriented programming, untrusted object deserialization

ACM Reference Format:
Joanna C. S. Santos, Xueling Zhang, and Mehdi Mirakhorli. 2022. Counterfeit Object-Oriented Programming Vulnerabilities: An Empirical Study in Java. In *Proceedings of the 1st International Workshop on Mining Software Repositories Applications for Privacy and Security (MSR4P&S '22), November 18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3549035.3561183

MSR4P&S '22, November 18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9457-4/22/11…$15.00
https://doi.org/10.1145/3549035.3561183

1 INTRODUCTION

Many modern applications, whether developed in Java, Python, PHP or other languages, rely on Object-Oriented (OO) design principles [5], where the basic system components are objects and classes. Many OO architectures [9] directly operate on objects; they share these objects with other processes [9, 17], store them in disk/-files for future retrieval [3] or transport them over network to other systems [9, 15]. The encapsulations provided by object structure, the concept of classes, and inheritance has increased programs reusability and extensibility [24]. Polymorphism has enabled separation of the client class from implementation code, and allows the object to decide which form of the function to implement at compile-time (overloading) as well as runtime (overriding).

Object-oriented programs also leverage numerous other dynamic features and design principles, which allow flexible design. They commonly use *runtime dispatching* to implement object polymorphism [7]. Dispatching is typically implemented using an *indirect function call*. Similarly, program constructs such as *reflection* allows an object-oriented program to modify its structure and behavior; other dynamic mechanisms such as *object-oriented callbacks* enable the application to handle subscribed events, arising at runtime, through a listener interface and respond using predefined concrete implementations. These features can be *abused* by the attackers to hijack the program's control flow to an undesirable behavior. This is referred to as ***Counterfeit Object-Oriented Programming*** (COOP) [23, 30]. COOP is a type of code reuse attack in which a hacker hijacks objects in the program (***gadgets***) and use them to control the program execution flow via manipulating the sequence of methods and data being passed among these methods (***gadget chains***).

Counterfeit object vulnerabilities are notoriously difficult to detect and even harder to prevent [30, 35]. Mainly because they do not exhibit the revealing characteristics of existing attack approaches, and exhibit control flow and data flow similar to those of benign code execution [27, 30]. An instance of such attack is *Deserialization of Untrusted Data* which is pervasive across Java applications, and it is also emerging in Python programs due to the use of object marshaling. Furthermore, there are numerous object-oriented programming approaches for transmitting, storing or extending the behavior of objects that can result in programs vulnerable to COOP attacks.

The literature had explored COOPs in lower-level languages such as C++ [23, 30], but these languages do not include metaprogramming features. Other languages (*e.g.*, Python, PHP, and Java) contain programming constructs (*e.g.*, native calls, reflection, and object serialization) which are used to load classes, invoke methods, instantiate objects and extend the programs' functionalities

at runtime. Although seemingly innocuous, these language constructs place the system at the risk of attackers tampering with objects (*gadgets*) in order to successfully execute code (*e.g.*, load a remote class, instantiate objects from it and execute its methods with a malicious purpose). Therefore, in this paper, we present a preliminary empirical study of COOP attacks in programs written in Java. We focused on COOP attacks caused by untrusted object deserialization.

In this study, we analyzed a total of 17 vulnerability reports caused by untrusted object deserialization. By collecting and analyzing several artifacts related to the problem (*e.g.*, released patch), we investigated the severity of COOP attacks caused by untrusted object deserialization (**RQ1**), what are their consequences (**RQ2**), and how developers mitigated the problem (**RQ3**).

We observed that these COOP attacks lead to vulnerabilities with a *high*/*critical* severity. Furthermore, the investigated attacks always resulted in *remote code execution*, where an attacker is able to craft an object in such a way they could invoke arbitrary methods and execute malicious commands. Finally, we found that developers mitigate the problem in three different ways: by preventing sensitive operations to be reachable from deserialization constructs, or by enforcing the integrity of deserialized objects, or by implementing compartmentalization.

The contributions of this paper are:

- A preliminary empirical investigation of COOP vulnerabilities caused by untrusted object deserialization.
- A discussion of their *consequences*, *severity*, and *mitigations*, which gives insights to developers on how they can avoid these vulnerabilities.
- A dataset [28] of COOP vulnerabilities caused by untrusted object deserialization.

This paper is organized as follows: Section 2 briefly describes COOP vulnerabilities to ensure that the essence of the paper can be understood by a broader audience. Section 3 describes our methodology in details. Section 4 presents the qualitative analysis of COOP vulnerability reports in order to identify their root causes and mitigations. Section 5 elaborates on threats to the validity of this work. Section 6 presents related work, and Section 7 concludes this paper, including planned future work.

2 BACKGROUND

To perform a COOP attack, attackers need to take control over one object in the application (the ***initial object***) [30]. The hijacking takes place by misusing a benign feature in a program that receives objects outside its trust boundary (*e.g.*, using a serialized object received from a socket, manipulating an object created by an external plug-in, *etc.*).

This initial object will have its fields initialized with attacker-controlled data. An object may contain other objects in its fields, creating potentially complex graph-like object layouts [8]. When the program later invokes one of its methods, it leads to a sequence of malicious method calls (***gadget chains***). The classes involved in a malicious method execution chain are referred to as ***gadget classes***.

On one hand, in lower level languages, the attack is performed by manipulating pointers in the program. For example, in C++, the attacker manipulates the *vtables* (virtual method tables) such that it triggers a sequence of method calls that result in a dangerous behavior [30]. In Java, on the other hand, attackers are not able to directly manipulate pointers and memory areas, instead, it would rely on objects already available in the classpath for use. Moreover, unlike C++, Java has reflection, a metaprogramming feature that allows classes to be loaded at runtime, and have their methods invoked; creating space for attackers to even be able to load remote classes (*i.e.*, outside the classpath).

Figure 1 contains three COOP attack scenario examples in Java. These attacks are caused by misusing three commonly used benign features: object deserialization, Remote Method Invocation (RMI), and the Java Naming and Directory Interface™ (JNDI). In these examples, consider that the classes in Listing 1 are available in the classpath. We explain these attacks in the next subsections.

2.1 Untrusted Object Deserialization

The first attack (Figure 1a) relies on *untrusted object deserialization*. Object serialization (also known as "marshaling") is a mechanism in which an object is converted to an abstract representation (*e.g.*, bytes, XML, JSON, *etc.*) that models the object's state (*i.e.*, fields' values and code). This abstract representation is suitable for network transportation, storage, and inter-process communication. The receiver of a serialized object has to parse the abstract representation in order to reconstruct a new object, a process called object deserialization (or "unmarshalling").

Although object serialization seems innocuous, several deserialization mechanisms allow arbitrary types to be deserialized and invoke methods from the objects' classes during their reconstruction (*e.g.*, default constructors, getter/setter methods, callback methods (also known as "magic methods", *etc.*) [19]. Attackers could leverage these methods invoked during object deserialization to conduct COOP attacks that can result in resource consumption (denial-of-service attacks), application crashes and remote code execution [8, 25].

The class `ObjectOutputStream` is part of Java's built-in deserialization API. It reconstructs an object from a *byte stream* that contains the object's fields values. This class can reconstruct any object, as long as its class implement the `java.io.Serializable` interface. If implemented by a `Serializable` class, the callback methods listed below are invoked by Java during deserialization. These methods, henceforth referred to as ***"magic methods"***, are the ones used by attackers to create a malicious sequence of method invocations (*gadget chain*):

(1) `void readObject(ObjectInputStream)`: it customizes the retrieval of an object's state from the stream.
(2) `void readObjectNoData()`: in the exceptional situation that a receiver has a subclass in its classpath but not its superclass, this method is invoked to initialize the object's state.
(3) `Object readResolve()`: this is the inverse of `writeResolve`. It allows classes to replace a specific instance that is being read from the stream.
(4) `void validateObject()`: it validates an object after it is deserialized. For this callback to be invoked, the class has to implement the `ObjectInputValidation` interface and register the

```java
class Task implements Runnable,
     Serializable {
  private String cmd;

  public CommandTask(String c) {
    this.cmd = c;
  }

  public void run() {    /* sink */
    Runtime.getRuntime().exec(cmd);
  }
}
```

```java
class TaskManager implements Serializable {
  private Runnable task;
  public TaskManager(Runnable t){ this.task = t; }
  private void readObject(ObjectInputStream ois){
    ois.defaultReadObject();
    task.run();
  }
}
interface Analyzer extends Remote {
  void analyze(Runnable r);
  void cleanResults();
}
```

```java
class AnalyzerImpl implements Analyzer{
  private File results;
  public AnalyzerImpl(File f){
    this.results = f;
  }
  public void analyze(Runnable r){
    r.run();
  }
  public void cleanResults(){
    results.delete(); /* sink */
  }
}
```

Listing 1: "Gadget classes" that can be used in a COOP attack to trigger a remote code execution.

```java
class IndexServlet extends HttpServlet {
  protected void doGet(HttpServletRequest rq,
       HttpServletResponse rs) {
    Cookie c = getCookieByName(rq, "user");
    if (c != null) {
      byte[] bytes = Base64.getDecoder()
        .decode(c.getValue());
      ObjectInput in = new ObjectInputStream(
        new ByteArrayInputStream(bytes)
      );
      User u = (User) in.readObject();
    } else { /* ... */ }
  }
}
```

(a)

```java
class RMIServer  {
  public static void main(String a[]){
    try {
      Analyzer obj = new AnalyzerImpl(null);
      Analyzer stub =
          (Analyzer) UnicastRemoteObject
          .exportObject(obj, 0);
      Registry registry =
          LocateRegistry.getRegistry();
      registry.bind("analyzer", stub);
    }catch (Exception e)
    { /* ... */ }
  }
}
```

(b)

```java
class JNDIExample{
  public static void main(String[]a){
    try {
      String name = a[0];
      Context ctx =
        new InitialContext();
      Analyzer analyzer =
        (Analyzer) ctx.lookup(name);
      analyzer.cleanResults();
    } catch (Exception e) {
      /* ... */
    }
  }
}
```

(c)

Figure 1: COOP attacks that rely on (a) untrusted object deserialization, (b) RMI, and (c) JNDI.

validator by invoking the method `registerValidation` from `ObjectInputStream` class.

As an example, Figure 1a contains a code snippet from a sample Web application (`IndexServlet`) that retrieves the "user" cookie from the HTTP request. This cookie is expected to contain a serialized `User` object encoded using *Base64*. An attacker could leverage the deserialization process to conduct a COOP attack by using two available serializable classes in the classpath (`TaskManager` and `Task` — gadget classes). An attacker would create a `TaskManager` object (`taskMgr`) as shown in Figure 2a. Then, the attacker serializes and encodes this malicious object in base64 and sends it as the "user" cookie to the Web application.

When the web application deserializes the object in the cookie, Java's deserialization mechanism (*ObjectInputStream*) invokes the callback method `readObject()` from the `TaskManager` class. It triggers the chain of method calls listed in Figure 2b. This gadget chain ends in a "sink"[1] – `exec()` – that executes a command to remove all files ("rm -rf /").

Although this request with a malicious serialized object will later trigger a `ClassCastException` (because the application attempt to cast the read object as a `User` type), the malicious command was already executed, because the type cast check occurs ***after*** the deserialization process took place.

2.2 RMI-based Attacks

The second example (`RMIServer` in Figure 1b) includes a sample Remote Method Invocation (RMI) server that exports an instance of the `AnalyzerImpl` class. An attacker can implement an RMI client that first looks up this object by its name ("analyzer") on the RMI server. Subsequently, this malicious client makes a remote procedure call to the `analyzer(Runnable r)` method passing as argument the malicious object `task` (in Figure 2a). This triggers the execution shown in Figure 2c which will lead to a recursive deletion of files in the root directory.

2.3 JNDI-based Attacks

In the third example (`JNDIExample` in Figure 1c), an application performs an object lookup by name using Java Naming and Directory Interface™ (JNDI) [32]. An attacker can implement a malicious RMI server that export the `analyzer` object in Figure 2a and binds it to the name "exploit". Subsequently, the attacker can invoke the program making a lookup to "rmi:/exploit", which will inject the malicious object and execute the call chain in Figure 2d, resulting in a deletion of the root directory.

3 METHODOLOGY

In this study, we focused on investigating COOP attacks caused by *untrusted object deserialization* (described in Section 2.1). In this section, we first introduce our research questions (Section 3.1), then we explain the methodology we followed to answer each of them (Section 3.2), and finally, we discuss how we compile our artifacts as a dataset (Section 3.3).

3.1 Research Questions

We answered the following research questions in this paper:

[1]Sinks are methods in the program's scope that performs sensitive operations, such as executing commands and manipulating file

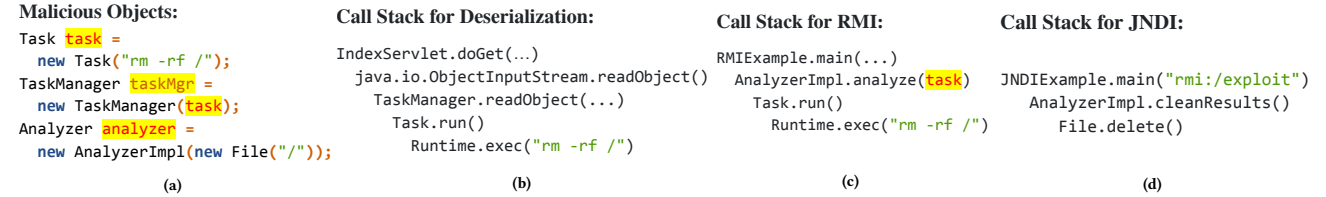

Figure 2: (a) Malicious objects crafted by an attacker. Call stacks for a successful COOP attack that relied on (b) object deserialization, (c) RMI, and (d) JNDI.

RQ1 *How severe are COOP attacks caused by untrusted object deserialization?*

We focused on understanding what is the perceived severity of these problems by developers.

RQ2 *What are the consequences of COOP attacks related to untrusted object deserialization vulnerabilities?*

We aimed to identify the faulty behavior observed when an untrusted object deserialization vulnerability is successfully executed.

RQ3 *How are COOP vulnerabilities related to untrusted object deserialization mitigated?*

We studied the strategies employed by developers to fix these vulnerabilities in real software systems.

3.2 Answering the Research Questions

To answer these questions, we conducted an in-depth analysis of vulnerability reports (CVEs) in the National Vulnerability Database (NVD). NVD is a well-known vulnerability database, which currently tracks over 191,000 vulnerabilities that exist in a variety of software products, both open and closed source.

Vulnerabilities disclosed in NVD are assigned a unique identifier known as "*CVE ID*" (Common Vulnerabilities and Exposure Identifier). Besides a CVE ID, each entry in NVD includes a short *description* of the problem and a list of *references*, *i.e.*, links to other Websites (such as issue tracking systems) that may contain more details about the CVE instance. NVD also indicates the software's *releases* affected by the vulnerability and a *severity score*.

Some CVE instances may also include *CWE tags* that indicate the *vulnerability type*. These tags are assigned by security analysts from the entities that reviewed the vulnerability report. The CWE tag refers to an entry from the *Common Weakness Enumeration (CWE)* dictionary [33], which enumerates common software/hardware weaknesses that may lead to vulnerabilities. A *weakness* denotes a family of security defects that share one or more aspect in common, such as a similar fault (*root cause*), failure (*consequence*), or fix (*repair*) [22]. Thus, the CWE tag is used by the NVD as a way to classify vulnerabilities.

Therefore, we first retrieved from NVD all the CVEs that either contained the keyword "`serializ`" in its *description* or whose *CWE tag* was equal to CWE-520 (Deserialization of Untrusted Data) [34]. Subsequently, we disregarded CVEs that (i) were in closed source systems, since there would not be enough public information for us to answer our research questions; or (ii) were in software systems implemented in a language other than Java.

We randomly selected a subset of 17 CVEs to identify its *severity*, *consequences*, and *mitigation techniques* implemented to fix the issue. Afterwards, we performed a qualitative analysis of these 17 CVEs and their associated artifacts to answer our research questions. We performed the following steps:

(1) For each CVE, we extracted its metadata from NVD (*description*, *CWE tag*, *references*, and *severity score*).
(2) We relied on the URLs in the references to identify the corresponding entry in the project's issue tracking system. From the issue tracking system entry, we then verify whether the vulnerability was acknowledged by developers and fixed. If a patch was publicly released, we collect both the project's vulnerable version and fixed version (that includes the fix).
(3) We manually analyzed these collected artifacts in order to capture information regarding the CVE's *vulnerable version* and *fixed version*, its *severity*, its *consequences*, as well as the *mitigation technique* implemented by developers to fix the problem. We obtained the severity for each CVE based on the CVSS score[2] provided by NVD. To identify the *consequences*, and *mitigation techniques*, we performed a qualitative analysis of the vulnerability report and associated artifacts. This qualitative analysis involved an open coding [21] in which we iteratively reviewed the artifacts and annotated each vulnerability with *codes*: one to indicate the mitigation technique used to fix the problem, and other(s) to indicate the consequence(s) of the vulnerability. During this open coding, we either annotated CVEs with codes already used or created new codes that emerged from the data (if the existing codes were not suitable for the CVE being analyzed). This open coding was performed by the first author, who has eight years of experience in software security.

After performing the above steps, we used the collected artifacts to answer each RQ as follows:

RQ1 We relied on the CVSS score provided by NVD, which is a number that ranges from 0 (least severe) to 10 (most severe).
RQ2 We answer this question by analyzing the consequences we observed while performing the open coding of CVEs.
RQ3 Similar to RQ2, this question is answered by inspecting the results of our open coding, in which we observed the different ways developers patched their projects.

[2]The Common Vulnerability Scoring System (CVSS) is a framework [20] used to measure the severity of a vulnerability.

3.3 A Dataset of COOP Vulnerabilities

After our qualitative analysis, we compiled these artifacts as a manually curated dataset of COOP vulnerabilities caused by untrusted object deserialization. This dataset includes a CSV file with the following metadata [28]: (i) **CVE ID**; (ii) the **vulnerable** and **fixed** versions of the project; (iii) **consequence**; (iv) CVSS score (**severity** – low, medium, high, critical); (v) **mitigation technique**.

4 RESULTS

In the next sections, we discuss our findings and answer our RQs.

4.1 RQ1: Severity

Table 1 presents the breakdown of the severity observed in the analyzed CVEs. The severity is based on the categorization given by the CVSS score v3. For two CVEs we analyzed[3], however, the severity score was based on CVSS v2 because there was no score provided using the version 3.x of the CVSS framework. The CVSS score ranges from 0 to 10, where a score from 0.1-3.9 is considered as *low* severity, 4.0-6.9 as *medium* severity, 7.0-8.9 as *high* severity, and 9.0-10.0 as *critical* severity.

Table 1: Severity of COOP vulnerabilities related to untrusted object deserialization

	Severity		
	Critical (9.0-10.0)	**High** (7.0-8.9)	**Medium** (4.0-6.9)
# CVEs	6 (35.2%)	9 (52.9%)	1 (5.8%)

We observe from the findings reported in Table 1 that the majority of vulnerabilities are classified as *high* severity. We also notice that 6 CVEs (35%) were also categorized as *critical* vulnerabilities. None of the vulnerabilities analyzed had a *low* severity score – the lowest observed CVSS score was 5.9 (*medium*) and the highest was 9.8 (*critical*).

One of the reasons as to why the severity scores were mostly high/critical was due to the fact that all the vulnerabilities had an attack vector through the network. That is, a hacker could deploy the attack remotely, making it easier to conduct successful attacks. This finding highlights the importance of studying COOP-related vulnerabilities.

4.2 RQ2: Consequences

We observed that all vulnerabilities lead to *remote code execution*. For one of the vulnerabilities (CVE-2016-1000031), besides code execution, an attacker could also manipulate local files (*e.g.*, delete/create local files).

The main attack vector used by intruders to execute arbitrary commands was via the use of *Java reflection*. That is, the gadget chain lead to a reflection construct that allowed attacks to load arbitrary classes, create instances, and invoke their methods using malicious data.

[3]These CVEs were: CVE-2015-6420 and CVE-2015-8103.

4.3 RQ3: Mitigation Techniques

By scrutinizing these 17 vulnerability reports, we observed that there were three ways that developers fixed COOP vulnerabilities caused by untrusted object deserialization: (i) by preventing untrusted data to reach a sink (***unreachable sinks***); (ii) by enforcing the integrity of serialized and deserialized objects (***enforcing integrity***); or (iii) ***compartmentalization***. The mitigation techniques and their corresponding category is presented in Table 2.

Table 2: Mitigation techniques for untrusted object deserialization

Category	Mitigation	#CVEs
Unreachable Sinks	M1.1 Allowed/blocked list of classes	7
	M1.2 Prevent deserialization of domain objects	4
	M1.3 Unsafe classes are no longer serializable	2
Enforcing Integrity	M2.1 Adding the "transient" to a sensitive field	1
	M2.2 Authenticate before deserializing an object	1
	M2.3 Replace Java's default deserialization API	2
Compartmentalization	M3.1 Deserialize within a sandbox	1

We can observe that developers mostly chose to fix vulnerabilities by making the sink unreachable. Among the mitigation strategies used to achieve this goal, the most used one was to create a list of classes that are allowed/blocked to be deserialized (**M1.1**). Some CVEs implemented multiple mitigations as part of their fix (*e.g.*, CVE-2015-6420 in Apache commons collections used mitigations **M1.2** and **M1.3**). In the next subsections, we elaborate on each of these mitigation techniques.

4.3.1 Group 1: Unreachable sinks. It contains mitigation techniques that make the sink *unreachable*. These mitigation techniques are:

M1.1 Allowed/Blocked list of classes: It maintains a list of classes that *may* or *may not* be deserialized (*allow list* and *block list*, respectively). When using Java's default deserialization API, this can be implemented by creating a subclass of `ObjectInputStream` that overrides the `resolveClass(ObjectStreamClass o)` method. This method throws an exception when the object type is either in the *block list* or not in the *allow list* [31].

Example: For instance, the CVE-2019-12384 is fixed by adding a gadget class into a list of *blocked classes* that cannot be deserialized, as shown below in the "unidiff" of the commit [13]. This commit blocks the serialization of instances of the class `DriverManagerConnectionSource`.

src/main/java/com/fasterxml/jackson/databind/jsontype/impl/SubTypeValidator.java

```
        // [databind#2326] (2.7.9.6): one more 3rd party gadget
        s.add("com.mysql.cj.jdbc.admin.MiniAdmin");
+
+       // [databind#2334] (2.9.9.1): logback-core
+       s.add("ch.qos.logback.core.db.DriverManagerConnectionSource");

        DEFAULT_NO_DESER_CLASS_NAMES = Collections.unmodifiableSet(s);
    }
```

We also observed that most of the fixes involved using a list of *"blocked classes"* (5 times) compared to the use of *"allow lists"*, which was observed in only one CVE. In another remaining vulnerability instance (CVE-2017-15693), we observed that it had a configuration mechanism that allowed users to create a list of blocked and allowed classes.

Although the use of *"blocked classes"* was the most common mitigation technique implemented, it is inherently problematic. New gadget classes, that are not in the list, can be found over time by attackers and be used to conduct malicious attacks. In fact, the CVE-2019-12384 with the fix shown above was due to not having a class from the logback core project in the malicious list.

One of the reasons as to why this occurs is because the use of blocked lists is easier to implement while minimizing the chances of backwards compatibility. The use of allow lists make the program more strict about what classes can be deserialized, making genuine program flows to be disrupted with the fix.

M1.2 Prevent deserialization of domain objects: This mitigation is typically used when the application has a class that extends another serializable class (directly or indirectly) which provides concrete implementations to callback methods (*i.e.*, "magic methods"). Therefore, to prevent malicious uses of these subclasses, the application breaks the chain of method calls by throwing an exception. Hence, the dangerous sink is unreachable because the chain of calls from a magic method – *e.g.*, `readObject(ObjectInputStream)` – to a sink method is broken due to a thrown exception.

Example: The jython project has a class named `PyFunction` that extends the class `PyObject`, which in turn implements the `java.io.Serializable` interface. This inheritance relationship makes the `PyFunction` class to be serializable too. In CVE-2016-4000, the `PyFunction` class was found to be used in successful COOP attacks. Hence, the fix implemented by developers prevents the class `Handler` to be deserialized. This is implemented by overriding the method `readResolve()` and making it throw an exception [1], as shown in the unidiff below for the fix:

src/org/python/core/PyFunction.java

```
    @Override
    public boolean isSequenceType() { return false; }

+   private Object readResolve() {
+       throw new UnsupportedOperationException();
+   }

    /* Traverseproc implementation */
    @Override
```

M1.3 Unsafe classes are no longer serializable: This mitigation technique involves making a gadget class no longer serializable. This is implemented by removing the "`extends Serializable`" from the class definition.

Example: In the Apache Commons FileUpload project version 1.3.2, a class named `DiskFileItem` implements the interface `FileItem`, which extends the `java.io.Serializable` interface. As a result, the `DiskFileItem` class also becomes serializable. In CVE-2016-1000031, researchers found that the `DiskFileItem` has a magic method (invoked during deserialization) that allowed an attacker to manipulate files. The project's developers fixed this problem by making `DiskFileItem` no longer serializable, as shown in the commit diff below [10]:

src/main/java/org/apache/commons/fileupload/FileItem.java

```
-public interface FileItem extends Serializable,FileItemHeadersSupport{
+public interface FileItem extends FileItemHeadersSupport {
```

4.3.2 Group 2: Enforcing object integrity. It encompasses the mitigation approaches below that enforce the integrity of the object:

M2.1 Add transient to a "sensitive" field: To prevent serializing fields with sensitive information (*e.g.*, passwords) or that are used as part of a gadget chain, applications enforce that these fields are not included when the object is serialized. This is achieved by adding the keyword `transient` to the field declaration [25]. By doing that, Java's built-in deserialization class ignores the field and does not write/read its value when serializing/deserializing the object.

Example: The beanshell project version 2.0b5 contains a serializable class named `XThis` that has a field named `invocationHandler`. This field is instantiated with a concrete implementation for the `java.lang.InvocationHandler` interface that uses reflection to invoke methods. An attacker relied on this class (`Handler`) to invoke arbitrary methods in the program (CVE-2016-2510). To fix this issue, the developers made the `Handler` class non-serializable (**M1.3**) and the `invocationHandler` field to be transient [11], as shown in the commit below:

src/bsh/XThis.java

```
-   InvocationHandler invocationHandler = new Handler();
+   transient InvocationHandler invocationHandler = new Handler();
...
-   class Handler implements InvocationHandler, java.io.Serializable
+   class Handler implements InvocationHandler
```

M2.2 Authenticate before deserializing an object: This mitigation is used when: (i) the application has to transmit objects, (ii) it does have a secure transport channel (*e.g.*, SSL) that can be used for authentication, and (iii) these objects need to be received in its entirety. In this case, marking fields as "transient" would not fulfill the application's needs [18]. This mitigation involves authenticating the remote source before receiving objects from it.

Example: In CVE-2016-3737 affecting the server in Red Hat JBoss Operations Network (JON) before 3.3.6, an attacker could craft a malicious object and send it to the server to trigger remote code execution. Since removing serialization and/or classes would not be a feasible mitigation, the fix for this issue involved manually configuring the JON to use SSL client authentication between servers and agents. The released version updated its documentation to guide the users on how to properly perform this configuration.

M2.3 Replace Java's default deserialization API: Java's built-in (de)serialization API allows arbitrary object types to be serialized/deserialized as long as it implements the `Serializable` interface. Since this API invokes methods from the objects' classes during their reconstruction (*i.e.*, magic methods), this built-in mechanism is deemed as inherently insecure [2]. Consequently, some applications decide to replace (or disable) this feature entirely to prevent vulnerabilities.

Example: In CVE-2017-1000034, the akka project disables Java's default serialization API and replaces it with its own (safer) serialization implementation [12].

4.3.3 Group 3: Compartmentalization. This category includes mitigation approaches in which the system enforces policies at runtime to prevent object deserialization misuse.

M3.1 Deserialize within a sandbox: A sandbox is used whenever an object is deserialized. This sandbox is configured with a set of policies that are enforced at runtime. Thus, if the deserialized object triggers an operation forbidden by the policy, the object reconstruction is stopped [14, 29]. Sandboxes are usually implemented using Java's `SecurityManager` class. This built-in class throws a `SecurityException` when it detects that a process is executing an operation not allowed by the security policy in place.
Example: To fix CVE-2018-1000058 (Jenkins project), developers made the deserialization of objects to be executed under a sandbox. Thus, an attacker is not able to execute arbitrary code in the pipeline. A (partial) implementation of the fix is shown in the code snippet below. The `SandboxedUnmarshaller` wraps the execution of all the deserialization operations such that they all run with sandbox protection.

org/jenkinsci/plugins/workflow/support/pickles/serialization/RiverReader.java

```
+ /** Applies {@link GroovySandbox} to a delegate unmarshaller. */
+ private static final class SandboxedUnmarshaller ... {
+
+   private final Unmarshaller delegate;
+
+   SandboxedUnmarshaller(Unmarshaller delegate) {
+     this.delegate = delegate;
+   }
+   ...
+
+   @Override public Object readObject() throws /* ... */ {
+     return sandbox(() -> delegate.readObject());
+   }
+
+   @Override public Object readObjectUnshared() throws /* ... */ {
+     return sandbox(() -> delegate.readObjectUnshared());
+   }
+   ...
+ }
```

4.4 Discussion

The key takeaways from our results are:

- **COOP attacks can lead to severe vulnerabilities**: This initial empirical study highlighted the importance of investigating COOP attacks. In our findings, we observed that CVEs related to untrusted object deserialization, a type of COOP attack, were often assigned by security analysts a high/critical severity score. One of the reasons being that attackers could deploy their attacks remotely, making it easier to reproduce attacks.
- **Developers may use inherently flawed/improper mitigations**: We observed that developers often used "blocked lists" (M1.1 discussed in Section 4.3) as a way to fix their vulnerability. The key problem, however, is that manually curating a list of dangerous classes lead to missing unknown gadget classes. That is, developers hardcode this list of dangerous classes based on prior knowledge of existing attacks. As new attacks are deployed, developers then have to patch the code by adding other class signatures to their list of blocked classes. This is a *reactive* mitigation strategy rather than a *proactive* approach.
- **There are trade-offs involved in the choice of employing a specific mitigation strategy**: We observed that there are multiple ways that developers fixed COOP vulnerabilities. The chosen mitigation strategy will often be a trade-off between the efforts required in changing the software, as well as backward compatibility considerations. As presented in Section 4.3, some mitigation strategies, such as replacing Java's default deserialization API (M2.3) would require extensive implementation and testing efforts. For that reason, developers often relied on a simpler solution, such as using a list of blocked classes that cannot be deserialized (M1.1). The use of "allow lists" is a safer alternative to the use of "blocked lists". However, this mitigation could also prevent the deserialization of genuine payloads, affecting the system's intended functionality. Therefore. although inherently less secure, the use of blocked lists was the most frequently employed strategy because it is easier to implement and reduce backward compatibility problems.

5 THREATS TO VALIDITY

One threat concerns the *construct validity* of our work; that is, to what extent the operational measurements we used are suitable for the purpose of our study [26]. In this context, two related threats are that (i) our analysis heavily depends on the accuracy of the collected reports (*i.e.*, CVEs, and patches, as described in Section 3.2) and (ii) the open coding of vulnerability reports. We mitigate this threat by following a *systematic process* in which we manually inspected each CVE and associated artifacts for completeness and accuracy. Moreover, this manual analysis was performed by one of the authors who has over 8 years of software security experience.

Another threat relates to the *generalizability* of the findings of the work (*external validity* [26]). We studied only the COOP attacks that are related to object deserialization and in Java programs. Since we analyzed a random sample that included only 17 vulnerabilities, we acknowledge the results may not generalize to other languages (*e.g.*, Python) and COOP attack types (*e.g.*, RMI-based COOP attacks). It is nonetheless important to highlight that our study's scope was not to find *generalizable* findings, but rather to give insights on this under-explored type of attacks and create a manually curated dataset that could help other researchers and practitioners.

6 RELATED WORK

The literature explored COOPs in lower-level languages such as C++ [4, 23, 30, 35], but these languages do not include metaprogramming features. Other languages (*e.g.*, Python, PHP, and Java) contain programming constructs (*e.g.*, native calls, reflection, and object serialization) which are used to load classes, invoke methods, create objects and extend the programs' functionalities at runtime. Although seemingly innocuous, these mechanisms place the system at the risk of attackers tampering with objects (*gadgets*) in order to successfully execute code (*e.g.*, load a remote class, instantiate objects from it and execute its methods with a malicious purpose).

Prior empirical studies explored vulnerabilities rooted in improper input validation problems, such as SQL injection, and buffer overflows [6, 16, 37] as well as language-specific vulnerabilities [36]. COOP vulnerabilities, however, are very different from these explored vulnerabilities. First, "dangerous operations" (*i.e.*, *sinks*) can be anywhere in the program's scope (*i.e.*, the language's built-in classes, library classes and the application code itself). Second, COOP attacks rely on dynamic programming features. Third, unlike these other classes of injection problems, in which the input

is a primitive or string, the input provided by the attacker is a specially crafted object. Hence, this study aimed to provide insights to developers on how to spot these problems and fix them.

7 CONCLUSION & FUTURE WORK

In this paper, we studied COOP attacks caused by untrusted object deserialization in Java programs. We investigated their severity, typical consequences, and mitigation techniques used by developers to prevent the attacks. Among our findings, we observed that deserialization-related COOP attacks were often flagged with a high severity. We also observed that one of the reasons for this high/critical severity was due to the fact that these attacks lead to remote code execution. We also found 7 different mitigation strategies employed by developers to prevent COOP attacks.

In the future, we plan to cover more vulnerabilities related to not only object deserialization, but also other COOP attack vectors (*e.g.*, RMI-based). Hence, we plan to (i) extract CVEs from NVD that are related to COOP, (ii) analyze publicly available exploits (iii) review the source code of open source systems with dynamic features such as *deserialization*, *RMI*, *JNDI*, *Dependency Injection*, *Java Management Extensions (JMX) API* and others that can enable counterfeit Object-Oriented programming attacks.

ACKNOWLEDGMENTS

This work was partially funded by the US National Science Foundation under grants number CNS-1816845 and CCF-1943300.

REFERENCES

[1] 2016. jython: d06e29d100c0. https://hg.python.org/jython/rev/d06e29d100c0 [Online; accessed 29. Jul. 2022].
[2] 2022. Secure Coding Guidelines for Java SE: Serialization and Deserialization. https://www.oracle.com/java/technologies/javase/seccodeguide.html#8 [Online; accessed 30. Jul. 2022].
[3] Tommi Aihkisalo and Tuomas Paaso. 2011. A Performance Comparison of Web Service Object Marshalling and Unmarshalling Solutions. In *2011 IEEE World Congress on Services*. 122–129. https://doi.org/10.1109/SERVICES.2011.61
[4] Markus Bauer and Christian Rossow. 2021. NoVT: Eliminating C++ Virtual Calls to Mitigate Vtable Hijacking. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 650–666.
[5] Grady Booch. 1982. Object-oriented design. *ACM SIGAda Ada Letters* 1, 3 (1982), 64–76.
[6] Larissa Braz, Enrico Fregnan, Gül Çalikli, and Alberto Bacchelli. 2021. Why Don't Developers Detect Improper Input Validation?'; DROP TABLE Papers;–. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 499–511.
[7] Brad Calder and Dirk Grunwald. 1994. Reducing Indirect Function Call Overhead in C++ Programs. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. ACM, New York, NY, USA, 397–408. https://doi.org/10.1145/174675.177973
[8] Jens Dietrich, Kamil Jezek, Shawn Rasheed, Amjed Tahir, and Alex Potanin. 2017. Evil Pickles: DoS Attacks Based on Object-Graph Engineering. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*, Vol. 74. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 10:1–10:32. https://doi.org/10.4230/LIPIcs.ECOOP.2017.10
[9] W. Emmerich and N. Kaveh. 2002. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002.* 691–692.
[10] GitHub. 2016. apache/commons-fileupload. https://github.com/apache/commons-fileupload/commit/02f6b2c4ef9aebf9cf8e55de8b90e73430b69385 [Online; accessed 30. Jul. 2022].
[11] GitHub. 2016. Avoid (de)serialization of XThis.Handler · beanshell/beanshell@1ccc66b. https://github.com/beanshell/beanshell/commit/1ccc66bb693d4e46a34a904db8eeff07808d2ced [Online; accessed 29. Jul. 2022].
[12] GitHub. 2017. akka/akka. https://github.com/akka/akka/commit/cc6561b47e5958923df520b8a9514010d3e11d49 [Online; accessed 30. Jul. 2022].
[13] GitHub. 2019. Fix #2334 · FasterXML/jackson-databind@c9ef4a1. https://github.com/FasterXML/jackson-databind/commit/c9ef4a10d6f6633cf470d6a469514b68fa2be234 [Online; accessed 28. Jul. 2022].
[14] GitHub. 2021. jenkinsci/workflow-support-plugin- Pipeline: Supporting APIs Plugin. https://github.com/jenkinsci/workflow-support-plugin/commit/a9b071025b5eea33176cefddc1928bce9904c0ef. [Accessed 07/17/2021].
[15] Konrad Grochowski, Michał Breiter, and Robert Nowak. 2019. Serialization in Object-Oriented Programming Languages. In *Introduction to Data Science and Machine Learning*, Keshav Sud, Pakize Erdogmus, and Seifedine Kadry (Eds.). IntechOpen, Rijeka, Chapter 12. https://doi.org/10.5772/intechopen.86917
[16] Munawar Hafiz and Ming Fang. 2016. Game of detections: how are security vulnerabilities discovered in the wild? *Empirical Software Engineering* 21, 5 (2016), 1920–1959.
[17] D. Hagimont and F. Boyer. 2001. A configurable RMI mechanism for sharing distributed Java objects. *IEEE Internet Computing* 5, 1 (2001), 36–43. https://doi.org/10.1109/4236.895140
[18] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. 2011. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional.
[19] Dustin Marx. 2018. JDK 11: Beginning of the End for Java Serialization? https://dzone.com/articles/jdk-11-beginning-of-the-end-for-java-serialization. (Accessed on 04/07/2020).
[20] P. Mell, K. Scarfone, and S. Romanosky. 2006. Common Vulnerability Scoring System. *IEEE Security Privacy* 4, 6 (Nov 2006), 85–89. https://doi.org/10.1109/MSP.2006.145
[21] Matthew B Miles, A Michael Huberman, and Johnny Saldana. 2013. *Qualitative data analysis*. Sage.
[22] Martin Monperrus. 2014. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 234–242.
[23] Paul Muntean, Richard Viehoever, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. 2021. ITOP: Automating Counterfeit Object-Oriented Programming Attacks. In *24th International Symposium on Research in Attacks, Intrusions and Defenses* (San Sebastian, Spain) *(RAID '21)*. ACM, New York, NY, USA, 162–176. https://doi.org/10.1145/3471621.3471847
[24] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. *SIGPLAN Not.* 51, 6 (jun 2016), 27–41. https://doi.org/10.1145/2980983.2908119
[25] Or Peles and Roee Hay. 2015. One Class to Rule Them All: 0-Day Deserialization Vulnerabilities in Android. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, Washington, D.C., 12 pages.
[26] Per Runeson and Martin Hoest. 2009. Guidelines for Conducting and Reporting Case Study Research in Software Engineering. *Empirical Software Engineering* 14 (2009), 131–164.
[27] Joanna C. S. Santos. 2021. *Understanding and Identifying Vulnerabilities Related to Architectural Security Tactics*. Ph. D. Dissertation. Rochester Institute of Technology.
[28] Joanna C. S. Santos, Xueling Zhang, and Mehdi Mirakhorli. 2022. *COOP Vulnerabilities Dataset*. https://github.com/SoftwareDesignLab/coop-dataset
[29] Will Sargent. 2021. Self-Protecting Sandbox using SecurityManager · Terse Systems. https://tersesystems.com/blog/2015/12/29/sandbox-experiment [Online; accessed 17. Jul. 2021].
[30] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. 2015. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications. In *2015 IEEE Symposium on Security and Privacy (S&P)*. IEEE, 745–762. https://doi.org/10.1109/SP.2015.51
[31] Robert Seacord. 2017. Combating Java Deserialization Vulnerabilities with Look-Ahead Object Input Streams (LAOIS).
[32] Michael Stepankin. 2019. Exploiting JNDI injections in Java. https://www.veracode.com/blog/research/exploiting-jndi-injections-java
[33] The MITRE Corporation 2022. *CWE - Common Weakness Enumeration*. The MITRE Corporation. http://cwe.mitre.org
[34] The MITRE Corporation 2022. *CWE-502: Deserialization of Untrusted Data*. The MITRE Corporation. http://cwe.mitre.org/data/definitions/502.html
[35] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. 2016. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 934–953.
[36] Morteza Verdi, Ashkan Sami, Jafar Akhondali, Foutse Khomh, Gias Uddin, and Alireza Karami Motlagh. 2020. An empirical study of C++ vulnerabilities in crowd-sourced code examples. *IEEE Transactions on Software Engineering* (2020).
[37] Tao Ye, Lingming Zhang, Linzhang Wang, and Xuandong Li. 2016. An empirical study on detecting and fixing buffer overflow bugs. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 91–101.