NCC Group Whitepaper

Jackson Deserialization Vulnerabilities

August 3, 2018 – Version 1.0

Prepared by
Robert C. Seacord – Technical Director

Abstract
The Jackson JSON processor offers an alternative to Java serialization by providing data binding capabilities to serialize Java objects to JSON and deserialize JSON back to Java objects. Poorly written Java code that deserializes JSON strings from untrusted sources can be vulnerable to a range of exploits including remote command execution (RCE), denial-of-service (DoS), and other attacks. These attacks are enabled by polymorphic type handling and deserialization to overly general superclasses. This white paper describes the features of Jackson serialization that makes it susceptible to exploitation, demonstrates a working exploit, and identifies effective mitigation strategies.

Table of Contents

1 Introduction

Deserialization of untrusted data has been proven to be almost universally dangerous regardless of language, platform, or serialization format. This class of vulnerability has been widely recognized by the security community for many years and is described by "CWE-502 Deserialization of Untrusted Data"[1] and OWASP vulnerability classifications Deserialization of Untrusted Data[2] and Object Injection.[3]

Java serialization was introduced in JDK 1.1 and is a Core Java feature [Ora14, GJS+16] that allows developers to transform a graph of Java objects into a stream of bytes for storage or transmission and then back into a graph of Java objects. Unfortunately, the Java serialization architecture is insecure which has led to numerous vulnerabilities including remote code execution (RCE) and denial-of-service (DoS) attacks [Sea17]. According to Oracle's Secure Coding Guidelines for Java SE [Ora17]:

> **Note: Deserialization of untrusted data is inherently dangerous and should be avoided.**
>
> Java Serialization provides an interface to classes that sidesteps the field access control mechanisms of the Java language. As a result, care must be taken when performing serialization and deserialization. Furthermore, deserialization of untrusted data should be avoided whenever possible, and should be performed carefully when it cannot be avoided.

As a result, developers have looked for alternative, secure solutions that provide the features and benefits of Java Serialization. One of the more popular alternatives is the Jackson JSON Processor.[4] Jackson provides a framework and data binding capabilities to serialize arbitrary Java objects to JSON and deserialize the JSON back to Java objects.[5] This is similar to Java Serialization in that Java programmers can serialize and deserialize *plain old Java objects* (POJO) and not just Java beans.

1.1 `ObjectMapper` Class

The primary class supporting Jackson Serialization is the `com.fasterxml.jackson.databind.ObjectMapper` class.[6] `ObjectMapper` is highly customizable enabling it to work with different styles of JSON content and to support advanced concepts such as polymorphism and object identity. The following code illustrates the simplest usage of `ObjectMapper` to serialize and deserialize an object instance to and from a file:

```
// Serialize then deserialize MyValue object instance from a file.
final ObjectMapper mapper = new ObjectMapper(); // can use static singleton
MyValue value = new MyValue();
File f = new File("serializationdata.json");
// Serialize MyValue instance to file
mapper.writeValue(f, value);
// Deserialize MyValue instance from file
MyValue value = mapper.readValue(f, MyValue.class);
```

Jackson is one of multiple Java libraries that can serialize Java object graphs into various representations (including YAML, AMF, JSON, XML, and various binary formats) and then restore the object graphs by deserialization. These libraries must interact with the source or target objects during serialization and deserialization to retrieve and set their properties. This interaction is most commonly based on the JavaBean's convention of accessing object properties through getter (`getXyz()`, possibly `isXyz()` for Boolean values)

[1] https://cwe.mitre.org/data/definitions/502.html
[2] https://www.owasp.org/index.php/Deserialization_of_untrusted_data
[3] https://www.owasp.org/index.php/PHP_Object_Injection
[4] https://github.com/FasterXML/jackson-docs
[5] http://www.studytrails.com/java/json/java-jackson-introduction/
[6] https://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/ObjectMapper.html

and setter methods (`setXyz()`). Other techniques include accessing the Java fields directly or by using the Java Reflection API [MM17, Bec17].

The expected root object type being deserialized is frequently known as the developer will want to do something with this object once it has been reconstructed. This can be used to recursively determine property types by using reflection.

1.2 Serialization Process

Jackson's serialization process is non-trivial and has evolved over time in an attempt to make it intuitive, or at least, less astonishing. Jackson automatically detects properties and determines how to serialize and deserialize them. The simplest way to ensure a field is both serializable and deserializable is to declare it `public` [Par17]. The existence of a getter method (that is, any gettable field with a matching name) makes non-`public` fields serializable. Unintuitively, any field with a getter is considered a property and consequently deserializable. Non-`public` fields may be accessed by non-`private` getters. A setter method only marks non-`public` fields as deserializable.

Jackson provides annotations that can make serialization and deserialization explicit. Annotations take precedence over automatic detection. Examples of annotations that influence serialization and deserialization include:

- `@JsonProperty` indicates that property is to be included.
- `@JsonAnySetter` defines a two-argument method as "any setter", used for deserializing values of otherwise unmapped JSON properties.
- `@JsonCreator` indicates that a constructor or static factory method should be used for creating value instances during deserialization.
- `@JsonSetter` is an alternative to `@JsonProperty` for marking methods as setters.

Additional Jackson annotations are documented on GitHub.[7]

Single argument constructors are automatically detected if the parameter type is `String`, `Boolean`, `boolean`, `Integer`, `int`, `Long`, or `long`.

`MapperFeature` controls many of the Jackson's automatic detection features. Refer to the documentation to view a list of customizable features and identify which are enabled by default.[8] Developers who prefer explicit settings typically disable the auto-detection settings found in `MapperFeature` and seldom, if ever, use default typing.

[7] https://github.com/FasterXML/jackson-annotations/wiki/Jackson-Annotations
[8] http://static.javadoc.io/com.fasterxml.jackson.core/jackson-databind/2.9.1/com/fasterxml/jackson/databind/MapperFeature.html

2 Polymorphic Type Handling

Deserialization must support polymorphic type handling to support Java inheritance and non-concrete types such as abstract classes and interfaces. The deserialization of these objects requires that appropriate type information is included in the object's representation so that they can be restored as the appropriate types [Sal17].

In the following example, serialization of `Person` instances is straightforward, as the types are available and can be introspected for their properties.

```
public class Person {
  public String name;
  public int age;
  public PhoneNumber phone; // embedded POJO
}
abstract class PhoneNumber {
   public int areaCode, local;
}
public class InternationalNumber extends PhoneNumber {
   public int countryCode;
}
public class DomesticNumber extends PhoneNumber { }
```

However, trying to restore these values generates an exception because `PhoneNumber` is an abstract type. Abstract classes cannot be instantiated and Jackson cannot determine which subtype to use without further information.

Polymorphic type handling is also required to correctly deserialize the following class hierarchy:

```
public class Zoo {
  public Animal animal;
}
static class Animal {  // All animals have names
  public String name;
  protected Animal() { }
}
static class Dog extends Animal {
  public double barkVolume; // in decibels
  public Dog() { }
}
static class Cat extends Animal {
  boolean likesCream;
  public int lives;
  public Cat() { }
}
```

Serialization of these objects must include sufficient type information to allow deserialization to instantiate the correct subtypes. In this example, deserialization can only statically determine that `Zoo` contains an `Animal`. During serialization, additional type information must be included for deserialization to determine if an `Animal` is a `Cat` or `Dog` instance.

2.1 Providing Type Information

Jackson provides multiple mechanisms for providing type information that can be used to enable polymorphic deserialization. The simplest of these mechanisms (and least secure) is to use default typing. Type information can be enabled for all objects via `ObjectMapper`:

```
ObjectMapper mapper = new ObjectMapper();
mapper.enableDefaultTyping(); // defaults to OBJECT_AND_NON_CONCRETE
mapper.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
```

The `enableDefaultTyping` method instructs `ObjectMapper` to include the Java class names for all non-`final` classes using the default inclusion mechanism of an additional wrapper array in JSON. The `ObjectMapper.DefaultTyping enum` can be used to limit the classes to which default typing is applied:

- `JAVA_LANG_OBJECT`: properties that have `Object` as the declared type (including generic types without an explicit type).
- `OBJECT_AND_NON_CONCRETE`: properties with the declared type of `Object` or an abstract type (abstract class or interface).
- `NON_CONCRETE_AND_ARRAYS`: all types covered by `OBJECT_AND_NON_CONCRETE` and arrays of these element types.
- `NON_FINAL`: all non-final types except for `String`, `Boolean`, `Integer`, and `Double` which can be correctly inferred from JSON; as well as for all arrays of non-`final` types.

Default typing is a security risk if used with untrusted content. A custom `TypeResolverBuilder` implementation can be used to limit possible types to instantiate:

```
setDefaultTyping(
  com.fasterxml.jackson.databind.jsontype.TypeResolverBuilder<?>)
);
```

Type information can also be provided using Jackson polymorphic type handling annotations:

- `@JsonTypeInfo` indicates details of what type information is included in serialization. This annotation can be used both for types (classes) and properties.
- `@JsonSubTypes` indicates sub-types of annotated type
- `@JsonTypeName` defines logical type name to use for annotated class

The previous `Zoo` example can be annotated as follows:

```
public class Zoo {
  public Animal animal;

  @JsonTypeInfo(
    use = JsonTypeInfo.Id.NAME,
    include = As.PROPERTY,
    property = "type")
  @JsonSubTypes({
    @JsonSubTypes.Type(value = Dog.class, name = "dog"),
    @JsonSubTypes.Type(value = Cat.class, name = "cat")
  )}

  public static class Animal {
```

```
    public String name;
  }

  @JsonTypeName("dog")
  public static class Dog extends Animal {
    public double barkVolume;
  }

  @JsonTypeName("cat")
  public static class Cat extends Animal {
    boolean likesCream;
    public int lives;
  }
}
```

These annotations allow the subclasses to be properly reconstructed. The `JsonTypeInfo.Id` specifies if the type identifier is the fully-qualified Java class name, the Java class name with minimal path, the logical name, uses a custom mechanism, or that no explicit type metadata is included and typing is purely done using contextual information, possibly augmented with other annotations. The `@JsonTypeInfo` for the `Animal` class specifies use of the logical name. The `JsonTypeInfo.As` enumeration specifies the inclusion mechanism. Several inclusion mechanisms are possible including as a single, configurable property or a wrapper array or wrapper object. Calling the `enableDefaultTyping` method is equivalent to the following annotation:

```
@JsonTypeInfo(use = Id.CLASS, include = As.WRAPPER_ARRAY)
```

One issue with this use of Jackson annotations is that it breaks encapsulation by having dependencies from the supertype to any possible subtype. This would make it difficult to use this mechanism, for example, when subclassing a class from a third party library. There is no way to address this problem with Jackson annotations, but it is also possible to register a specified class as a subtype using the `registerSubtypes` method in `ObjectMapper` or implementing a custom `TypeResolverBuilder`.

The ability to indicate all types is a core requirement for flexible deserialization. A significant portion of real usage defines `java.lang.Object`—allowing any class which can be resolved to be deserialized.

3 Vulnerabilities

Poorly written Java code that deserializes JSON strings from untrusted sources can be vulnerable to a range of exploits including remote command execution (RCE), denial-of-service (DoS), and other attacks. These attacks are possible when an attacker can control the contents of the JSON to specify the deserialization of dangerous objects that will invoke specific methods already present in the JVM with attacker-supplied data.

Security researchers have identified a wide-variety of existing classes that can be used to violate the security policies of a vulnerable Java program. These classes are often referred to as *gadgets* because they are similar to gadgets in return-oriented programming [Sha07]. Gadgets consist of existing, executable code present in the vulnerable process that can be maliciously repurposed by an attacker. In the case of Jackson deserialization vulnerabilities, these classes contain code that is executed when an object is deserialized.

The following is a partial list of known gadgets taken from the black list for Jackson deserialization[9]:

- `org.apache.commons.collections.functors.InvokerTransformer`
- `org.apache.commons.collections.functors.InstantiateTransformer`
- `org.apache.commons.collections4.functors.InvokerTransformer`
- `org.apache.commons.collections4.functors.InstantiateTransformer`
- `org.codehaus.groovy.runtime.ConvertedClosure`
- `org.codehaus.groovy.runtime.MethodClosure`
- `org.springframework.beans.factory.ObjectFactory`
- `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl`

3.1 Sample Exploit

A sample exploit, available on GitHub[10] was developed to demonstrate the use of the `com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl` gadget.

The Xalan-Java Compiling processor, XSLTC, can be used to compile the following source code into bytecodes for a `translet` object within the TemplatesImpl instance:

```
java.lang.Runtime.getRuntime().exec(new String[] { "Calc.exe" });
```

This gadget is encoded in the following JSON string:

```
["org.apache.xalan.xsltc.trax.TemplatesImpl",
  {
    "transletBytecodes":["…yv66vg"],
    "transletName":"foo",
    "outputProperties":{}
  }
]
```

Note that the bytecodes and not the actual class definitions are stored when serializing a `TemplatesImpl` object.

The `transletBytecodes` are executed when the `getOutputProperties` method is called by reflection during deserialization to read the `outputProperties` property as can be seen in the following stack trace:

[9] https://github.com/FasterXML/jackson-databind/commit/fd8dec2c7fab8b4b4bd60502a0f1d63ec23c24da
[10] https://github.com/rcseacord/JavaSCR/tree/master/JavaSCR/src/jackson

```
<init>:56, Pwner60092316258519 (ysoserial)
newInstance0:-1, NativeConstructorAccessorImpl (sun.reflect)
newInstance:62, NativeConstructorAccessorImpl (sun.reflect)
newInstance:45, DelegatingConstructorAccessorImpl (sun.reflect)
newInstance:422, Constructor (java.lang.reflect)
newInstance:442, Class (java.lang)
getTransletInstance:340, TemplatesImpl (org.apache.xalan.xsltc.trax)
newTransformer:369, TemplatesImpl (org.apache.xalan.xsltc.trax)
getOutputProperties:390, TemplatesImpl (org.apache.xalan.xsltc.trax)
invoke0:-1, NativeMethodAccessorImpl (sun.reflect)
invoke:62, NativeMethodAccessorImpl (sun.reflect)
invoke:43, DelegatingMethodAccessorImpl (sun.reflect)
invoke:497, Method (java.lang.reflect)
deserializeAndSet:105, SetterlessProperty (com.fasterxml.jackson.databind.deser.impl)
vanillaDeserialize:260, BeanDeserializer (com.fasterxml.jackson.databind.deser)
deserialize:125, BeanDeserializer (com.fasterxml.jackson.databind.deser)
_deserialize:110, AsArrayTypeDeserializer (com.fasterxml.jackson.databind.jsontype.
    impl)
deserializeTypedFromAny:68, AsArrayTypeDeserializer (com.fasterxml.jackson.databind.
    jsontype.impl)
deserializeWithType:554, UntypedObjectDeserializer$Vanilla (com.fasterxml.jackson.
    databind.deser.std)
deserialize:42, TypeWrappedDeserializer (com.fasterxml.jackson.databind.deser.impl)
_readMapAndClose:3789, ObjectMapper (com.fasterxml.jackson.databind)
readValue:2779, ObjectMapper (com.fasterxml.jackson.databind)
deserialize:46, Cage (jackson)
main:121, Cage (jackson)
```

This exploit is interesting in that it makes use of Java code already present in the JVM (Xalan) and injected code (the `exec` call). The exploit is also interesting in that it utilizes a getter and not a setter method.

The deserialization vulnerability exploited in this example occurs in jackson-databind versions before 2.6.7.1, 2.7.9.1 and 2.8.9 and is described by CVE-2017-7525[11] and tracked as "Jackson Deserializer security vulnerability via default typing (CVE-2017-7525) #1599".[12] The sample exploit was tested with `Jackson-databind-2.7.3`. The exploit was successfully executed against both:

- JRE Xalan (`com.sun.org.apache.xalan`) using JDK 8u40
- Apache Xalan (`org.apache.xalan`) using the latest release available at the time (Xalan-Java Version 2.7.1)

[11] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-7525
[12] https://github.com/FasterXML/jackson-databind/issues/1599

4 Mitigations

The primary mitigation for Jackson deserialization vulnerabilities is to never deserialize untrusted data. The deserialization of untrusted data is inherently unsafe because the attacker can tamper with the data. In the exploit detailed in this paper, the attacker exploits this to remotely execute arbitrary commands on the vulnerable system. However, the attacker may also simply create objects with invalid invariants that may be used to corrupt the system data or to circumvent security controls.

Jackson has added a blacklist of known gadget types to mitigate the risk from exploits using these gadgets[13,].[14] Blacklists are not a particularly effective solution and can be best thought of as a stopgap measure. The blacklist is only effective if the set of all dangerous gadgets can be enumerated. This is clearly not the case as can be seen in the following table of CVE details[15]:

#	CVE ID	CWE ID	Vulnerability Type(s)	Publish Date	Update Date	Score	Gained Access Level	Access	Complexity	Authentication
1	CVE-2018-7489	502	Exec Code Bypass	2018-02-26	2018-06-28	7.5	None	Remote	Low	Not required
FasterXML jackson-databind before 2.8.11.1 and 2.9.x before 2.9.5 allows unauthenticated remote code execution because of an incomplete fix for the CVE-2017-7525 deserialization flaw. This is exploitable by sending maliciously crafted JSON input to the readValue method of the ObjectMapper, bypassing a blacklist that is ineffective if the c3p0 libraries are available in the classpath.										
2	CVE-2018-5968	502	Exec Code Bypass	2018-01-21	2018-05-16	5.1	None	Remote	High	Not required
FasterXML jackson-databind through 2.8.11 and 2.9.x through 2.9.3 allows unauthenticated remote code execution because of an incomplete fix for the CVE-2017-7525 and CVE-2017-17485 deserialization flaws. This is exploitable via two different gadgets that bypass a blacklist.										
3	CVE-2017-17485	94	Exec Code Bypass	2018-01-10	2018-05-16	7.5	None	Remote	Low	Not required
FasterXML jackson-databind through 2.8.10 and 2.9.x through 2.9.3 allows unauthenticated remote code execution because of an incomplete fix for the CVE-2017-7525 deserialization flaw. This is exploitable by sending maliciously crafted JSON input to the readValue method of the ObjectMapper, bypassing a blacklist that is ineffective if the Spring libraries are available in the classpath.										
4	CVE-2017-15095	502	Exec Code	2018-02-06	2018-05-16	7.5	None	Remote	Low	Not required
A deserialization flaw was discovered in the jackson-databind in versions before 2.8.10 and 2.9.1, which could allow an unauthenticated user to perform code execution by sending the maliciously crafted input to the readValue method of the ObjectMapper. This issue extends the previous flaw CVE-2017-7525 by blacklisting more classes that could be used maliciously.										
5	CVE-2017-7525	502	Exec Code	2018-02-06	2018-05-16	7.5	None	Remote	Low	Not required
A deserialization flaw was discovered in the jackson-databind, versions before 2.6.7.1, 2.7.9.1 and 2.8.9, which could allow an unauthenticated user to perform code execution by sending the maliciously crafted input to the readValue method of the ObjectMapper.										

The last entry in the table for CVE-2017-7525 describes the original vulnerability while later CVEs are the result of gadget types that were omitted, and consequently bypass, the blacklist mechanism. Furthermore, it would be foolish to think that the current blacklist is complete as gadgets discovered by attackers are seldom reported.

[13] https://github.com/FasterXML/jackson-databind/issues/1599
[14] https://github.com/FasterXML/jackson-databind/commit/fd8dec2c7fab8b4b4bd60502a0f1d63ec23c24da
[15] https://www.cvedetails.com/vulnerability-list/vendor_id-15866/product_id-42991/Fasterxml-Jackson-databind.html

Besides not deserializing untrusted data, the most effective mitigation of the risk from Jackson deserialization vulnerabilities is to eliminate the two conditions which allow it. The first of these is polymorphic type handling using the names of the classes to identify the classes. The most dangerous example of this is a call to the `enableDefaultTyping` method which instructs `ObjectMapper` to include the Java class names for all non-`final` classes. When polymorphic type handling is required, it is best to consider it to be a privilege and following the principle of least privilege [SS75]. Polymorphic type handling should only be enabled for types that require this mechanism for proper deserialization. It can be implemented using a custom `TypeResolverBuilder` implementation or the use of Jackson polymorphic type handling annotations. It is also important to specify the type using the logical name and not the class name, for example: `@JsonTypeInfo(use = Id.NAME)`.

The second issue is overly permissive deserialization types. Unfortunately, it is common practice to allow any objects to be deserialized, as in the following example:

```
private static Object deserialize(String data) throws IOException {
  ObjectMapper mapper = new ObjectMapper();
  mapper.enableDefaultTyping();
  // deserialize as Object or permissive tag interfaces such as java.io.Serializable
  return mapper.readValue(data, Object.class);
}
```

Interface types such as `java.util.Serializable` are also dangerous because these types are commonly implemented by classes that may be used as gadgets.

Instead of specifying `Object.class`, the developer should specify the specific type of the expected root object. These objects must be carefully written to ensure that they cannot be used as a gadget in an exploit.

Jackson allows various features of deserialization to be customized, including a number of features to enable/disable failure modes. Enabling all failure modes will provide additional input validation and should help improve security. In particularly, do not disable the `FAIL_ON_UNKNOWN_PROPERTIES` feature (which is enabled by default):

```
mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
```

Disabling this feature typically masks a wide range of problems.

5 Conclusion

Poorly written Java code that deserializes JSON strings from untrusted sources can be vulnerable to a range of exploits including remote command execution (RCE), denial-of-service (DoS), and other attacks. These attacks are possible under two conditions. First, the Java class name (either the fully-qualified or minimal path) is included with the JSON to identify the object type during deserialization. Second, the resulting type of the deserialized object has a subclass, or an interface implementation that can be used as a *gadget*.

Most serialization libraries are susceptible to remote command execution and other exploits when deserializing untrusted serialized data. Jackson does not perform typing by default (including collection generic types) and does not allow the specification of arbitrary types. Unfortunately, default typing is necessary to correctly deserialize Java subclasses and it is common practice to for developers to specify `java.lang.Object.class` to allow any object to be serialized and deserialized. The combination of polymorphic type handling and deserialization to overly general superclasses allows attackers to provide specially-crafted JSON which can be used in a wide-variety of attacks including, but not limited to, remote command execution.

6 Acknowledgments

Thanks to Jeremy Brandt-Young, Gene Meltser, David Goldsmith, Andy Grant, and Clint Gibler for supporting this effort. Thanks to the technical reviewers Peter Greko, Paul Tetreau, Tatu Saloranta, Moritz Bechler, and Fred Long (Aberystwyth University).

7 Author Bio

Robert C. Seacord is a Technical Director with NCC Group where he works with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. Previously, Robert led the secure coding initiative in the CERT Division of Carnegie Mellon University's Software Engineering Institute (SEI). Robert is also an adjunct professor in the School of Computer Science and the Information Networking Institute at Carnegie Mellon University.

Robert is the author of six books, including *The CERT C Coding Standard, Second Edition* (Addison-Wesley, 2014), *Secure Coding in C and C++*, Second Edition (Addison-Wesley, 2013), *The CERT Oracle Secure Coding Standard for Java* (Addison-Wesley, 2012) [LMS+11], and *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (Addison-Wesley, 2014) [LMS+13]. Robert is on the Advisory Board for the Linux Foundation and an expert on the ISO/IEC JTC1/SC22/WG14 international standardization working group for the C programming language.

References

[Bec17] Moritz Bechler. Java unmarshaller security: Turning your data into code execution. https://github.com/mbechler/marshalsec/blob/master/marshalsec.pdf, May 22, 2017. 4

[GJS+16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. Java language specification: Java se 8 edition. https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf, 2016. 3

[LMS+11] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. *The CERT Oracle Secure Coding Standard for Java*. Addison-Wesley Professional, 2011. 14

[LMS+13] Fred Long, Dhruv Mohindra, Robert C Seacord, Dean F Sutherland, and David Svoboda. *Java coding guidelines: 75 recommendations for reliable and secure programs*. Addison-Wesley, 2013. 14

[MM17] Alvaro Muñoz and Oleksandr Mirosh. Friday the 13th: Json attacks. https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-JSON-Attacks-wp.pdf, July 2017. 4

[Ora14] Oracle. Java platform, standard edition 8 api specification. https://docs.oracle.com/javase/8/docs/api/, 2014. 3

[Ora17] Oracle. Secure coding guidelines for java se, version 6.0 updated for java se 9. http://www.oracle.com/technetwork/java/seccodeguide-139067.html, September 28, 2017. 3

[Par17] Eugen Paraschiv. Jackson - decide what fields get serialized/deserialized. http://www.baeldung.com/jackson-field-serializable-deserializable-or-not, July 20, 2017. 4

[Sal17] Tatu Saloranta. On jackson cves: Don't panic—here is what you need to know. https://medium.com/@cowtowncoder/on-jackson-cves-dont-panic-here-is-what-you-need-to-know-54cd0d6e8062, December 22, 2017. 5

[Sea17] Robert Seacord. Combating java deserialization vulnerabilities with look-ahead object input streams (laois). https://www.nccgroup.trust/us/our-research/combating-java-deserialization-vulnerabilities-with-look-ahead-object-input-streams-laois/, 2017. 3

[Sha07] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007. 8

[SS75] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975. 11