Deserialization of untrusted data in Java

Analysis, current solutions & a new approach

Apostolos Giannakidis
@apgiannakidis

OWASP London Meetup
18th May 2017

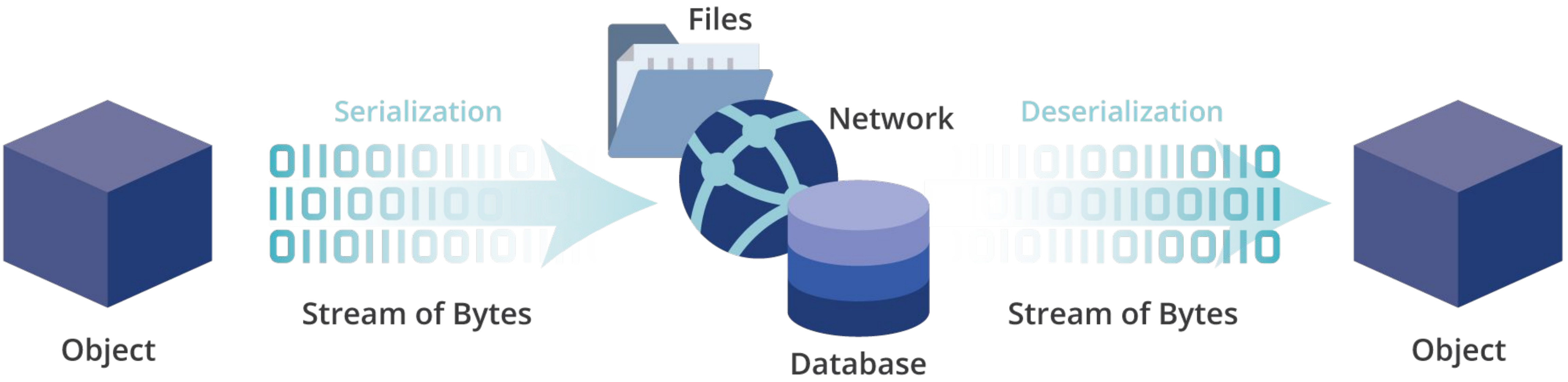
Whois

- Security Architect at Waratek
- Application security
- Vulnerability and exploit research
- R&D exploit mitigation
- Product development
- Over a decade of professional experience in software and security
- MSc Computer Science

Agenda

- Java serialization basics
- Deserialization of untrusted data
- Understanding the vulnerability and the exploits
- Common misconceptions
- Known mitigations and their limitations
- A new mitigation approach using runtime virtualization
- Q & A

Serialization 101



Use Cases

- Remote / Interprocess Communication (RPC/IPC)
- Message Brokers
- Caching
- Tokens / Cookies
- RMI
- JMX
- JMS

Serialization Format

- Data only
- Class metadata
  - Names of data types
  - Names of object fields
- Object field values

Serializable is not easy

”Allowing a class’s instances to be serializable can be as simple as adding the words “implements Serializable” to the class.

*This is a common misconception, **the truth is far more complex.**”*

- Joshua Bloch

Effective Java

Serializable makes objects untrusted

- Serializable creates:
  - a **public** hidden constructor
  - a **public** interface to all fields of that class
- Deserialization is **Object Creation** and **Initialization**
  - Without invoking the actual class's constructor
- Treat it as a **Constructor**
  - Apply same input validation, invariant constraints, and security permissions
  - Before any of its methods is invoked!

Serializable is a commitment

- Audit your Serializable classes
- Create a Threat Model
- Class definitions evolve
  - Re-evaluate threat models on every new class version
- Document all deserialization end-points

Attacking Java Serialization

Focus on attack techniques found by Gabriel Lawrence, Chris Frohoff, Steve Breen, Matthias Kaiser, Alvaro Muñoz

- Integrity
  - RCE via gadget chains
- Availability
  - DoS via gadget chains

Misconception #1

My app does not use serialization, so I am safe

- Custom Java App
- 3rd party libs (Apache Commons, Spring, Log4j, etc.)
- Middleware (IBM WebSphereMQ, Oracle OpenMQ, Apache ActiveMQ, JBoss EAP, etc.)
- App Server (Oracle WebLogic, IBM WebSphere, etc.)

Who is affected?

- Oracle
- Red Hat
- Apache
- IBM
- Symantec
- VMWare
- Cisco
- Pivotal
- Atlassian
- Jenkins

Virtually everyone!

Deserialization of untrusted data (CWE-502)

```
InputStream untrusted = request.getInputStream();
ObjectInputStream ois = new ObjectInputStream( untrusted );
SomeObject deserialized = (SomeObject) ois.readObject();
```

- What is the problem here?
- **Any** available class can be deserialized
- Calling ObjectInputStream.readObject() using **untrusted** data can result in malicious behavior
  - Arbitrary code execution
  - Denial of Service
  - Remote command execution
    - Malware / Ransomware infection

SFMTA Ransomware Incident

- San Francisco Municipal Transportation Agency
- Ransomware infection via Java Deserialization RCE
- ~ 900 computers
- $559k in fares daily loss
- Exfiltrated 30GB of files

Source: https://www.thesslstore.com, https://arstechnica.com

Misconception #2

I am deserializing trusted data, so I am safe

- What is trusted data?
- Sources that are trusted today may not be tomorrow

Abusing Java Deserialization

- Attackers find dangerous classes *available* in the system
  - Not necessarily *used* by the system
- *Dangerous* classes (NOT necessarily vulnerable)
  - extend Serializable or Externalizable
  - utilize their member fields during or after deserialization
  - no input validation
- Known as **gadget** classes
  - JRE, App Servers, common libraries, frameworks, Apps
  - e.g., Apache Commons Collections InvokerTransformer

Misconception #3

ACC InvokerTransformer is on my ClassPath, therefore I am vulnerable

- **Not** a vulnerability of the ACC InvokerTransformer
- The vulnerability is the deserialization of untrusted data
- The InvokerTransformer simply made the vulnerability **exploitable**

Unrealistic Gadget

```
public class SomeClass implements Serializable {
  private String cmd;
  private void readObject( ObjectInputStream stream )
      throws Exception {
        stream.defaultReadObject();
        Runtime.getRuntime().exec( cmd );
  }
}
```

Unrealistic Gadget

```
public class SomeClass implemen
  private String cmd;
  private void readObj          utStr
      throws Exception {
        stream.defaultReadObject
        Runtime.getRuntime().exec( cmd );
  }
}
```

Remote Shell

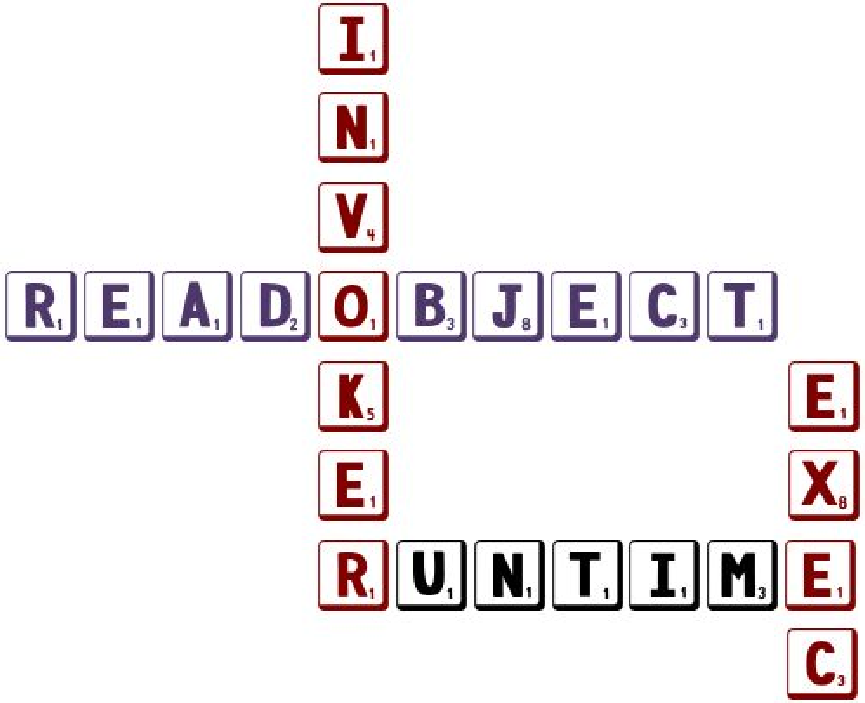
By Design!

Chaining Gadgets together

- Attackers create chains of method calls
  - Known as **gadget chains**
  - Abuse the deserialization logic
- Gadget Chains are self-executing
  - Triggered by the JVM during or after deserialization
  - Their goal is to exhibit malicious behavior

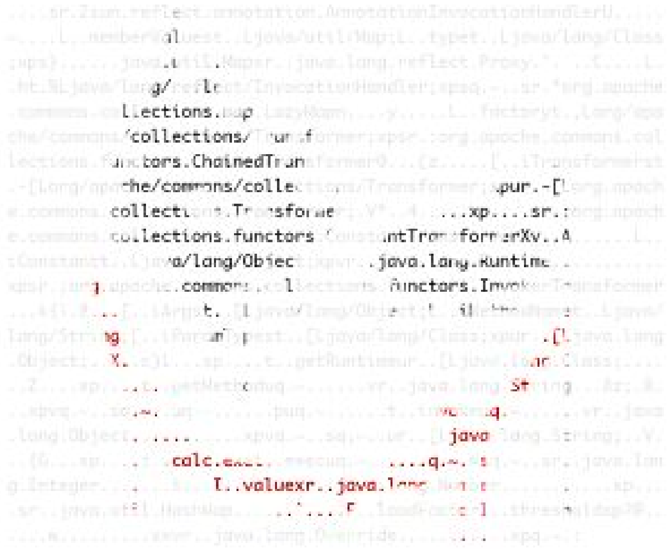
Gadget Chain Creation

- Gadget chain creation is like a game of Scrabble
- Gadgets are letters of the words
- Gadget chains are words
  - correct words win the game
- The more classes you have loaded
  - the more letters you have
  - more chances to create words
  - more likely to be exploitable



Do It Yourself

- Ysoserial, by Chris Frohoff
- PoC payload generation tool
- Tens of ready-to-use gadgets
- https://github.com/frohoff/ysoserial/



Possible Mitigations

- Avoid object serialization
- WAFs / Firewalls
- Custom Java Security Manager
- Filter trusted / untrusted classes
  - Blacklisting
  - Whitelisting

Existing Mitigations

Avoid Object Serialization

- Recommended
- Redesign / re-architect the software
- But you may still be vulnerable
- Deserialization may still occur in components you don’t control

Existing Mitigations

WAFs / Firewalls

- Block ports and apply basic heuristics
- Can produce false positives
- Lack visibility of the runtime
- Runtime provides full context
- Protection should be in the runtime

Existing Mitigations

Checking WAFs for False Positives

```
HashMap<String, String> map = new HashMap<>();
map.put( “org.apache.commons.collections.functors.InvokerTransformer”,
            “calc.exe” );
FileOutputStream file = new FileOutputStream( "out.bin" );
ObjectOutputStream out = new ObjectOutputStream(file);
out.writeObject( map );
out.close();
```

Existing Mitigations

Filter Untrusted Classes - Blacklisting

- Always a bad idea
- Never complete
- False sense of security
- Requires profiling
- Not possible if gadget class is needed
- Can be bypassed (see A.Muñoz & C.Schneider Serial Killer: Silently Pwning Your Java Endpoints)

Filter Trusted Classes - Whitelisting

- Better approach than Blacklisting
- Requires profiling
- Difficult to configure
- No protection if gadget class is needed
- May not protect against Golden Gadgets
    - SerialDoS
    - SerialDNSDoS
    - <= JRE 1.7u21
    - Many more…

Existing Mitigations

Maintaining lists is a commitment

- Whitelists may need to be updated on new releases
- Blacklists must be updated on every new gadget
- Forgetting to whitelist a class breaks your app
- Forgetting to blacklist a class makes you vulnerable

Risk-based Management using whitelists

- Who should be responsible for their maintenance?
- Difficult to apply risk-based management
  - How should a class’s risk profile be assessed?
  - Devs understand code
  - Security teams understand operations

Existing Mitigations

Whitelisting is not easy

- Dev asks Security team to whitelist a new class: SomeClass

```
class SomeClass extends BaseClass {
    // nothing suspicious
}
```

- Security team whitelists the class

Whitelisting is not easy

- Dev asks Security team to whitelist a new class: SomeClass

```
class SomeClass extends BaseClass {
    // nothing suspicious
}
```

- Security team whitelists the class

```
class BaseClass extends HashMap {
}
```

- Vulnerable to SerialDoS

JEP 290 - Serialization Filtering

- White / Black listing approach
- 3 types of filters
  - Global Filter
  - Specific Filter
  - Built-in Filters
- Graph and Stream Limits
- Patterns to whitelist classes and package

Existing Mitigations

Custom Java Security Manager

- Always a good idea
- It’s a type of whitelisting
- Requires profiling
- Difficult to configure
- Can be bypassed
  - Deserialization payload can unset the Security Manager
  - See ZoneInfo Exploit (CVE-2008-5353)
- Does not protect against some DoS attacks
- Does not protect against deferred attacks (such as finalize())

Apache Commons Collections Gadget Chain

```
ObjectInputStream.readObject()
        AnnotationInvocationHandler.readObject()
            Map(Proxy).entrySet()
                AnnotationInvocationHandler.invoke()
                    LazyMap.get()
                        ChainedTransformer.transform()
                                   ...
                                Method.invoke()
                                    Runtime.getRuntime()
                            InvokerTransformer.transform()
                                Method.invoke()
                                    Runtime.exec()
```

Source: Chris Frohoff
Marshalling Pickles
AppSecCali 2015

JRE 1.7u21 Gadget Chain

```
LinkedHashSet.readObject()
  ...
  LinkedHashSet.add()
      ...
      Proxy(Templates).equals()
            ...
            ClassLoader.defineClass()
                        Class.newInstance()
                              ...
                                 Runtime.exec()
```

Source: Chris Frohoff
ysoserial

Let’s revisit the core of the problem

- The JVM is *irrationally* too permissive
- Does not protect against API Abuse & Privilege Escalation
  - It is not even safeguarding its own invariants!
- The JVM makes zero effort to mitigate attacks
- Asking developers to “*just write better code*” is not the answer

Let’s revisit the core of the problem

The runtime platform does not provide a secure execution environment by default

What do the Standards suggest?

CERT Secure Coding Standards

- SER08-J. Minimize privileges before deserializing from a privileged context
- SEC58-J. Deserialization methods should not perform potentially dangerous operations

MITRE

- CWE-250: Execution with Unnecessary Privileges
  - [...] isolate the privileged code as much as possible from other code. Raise privileges as late as possible, and drop them as soon as possible.
- CWE-273: Improper Check for Dropped Privileges
  - Compartmentalize the system to have "safe" areas where trust boundaries can be unambiguously drawn.

New Mitigation Approach

Runtime Micro-Compartmentalization

- Defines boundaries around operations
- Controlled communication between compartments
- Nested micro-compartments
- Fine-grained visibility
- Activated:
  - during deserialization
  - on method invocations of deserialized objects
    - such as finalize()

New Mitigation Approach

Runtime Virtualization

- If runtime protections share address-space/name-space with an untrusted App then the runtime protection also cannot be trusted
- Virtualization is the only proven way for trusted software (e.g. a hypervisor) to quarantine and control untrusted software
- Enforces isolation and contextual access control
- Untrusted data are tracked at runtime via - always on - memory tainting

New Mitigation Approach

Runtime Privilege De-Escalation

- Compartments drops specific sets of privileges
  - Privileges are API calls, arguments, exceptions, etc
  - Principle of least privilege could also be applied
- Compartments sets sensible resource limits
- Prohibits mutation of the JVM’s state
- Prohibits tainted I/O to exit the JVM
- Maintains JVM invariants

New Mitigation Approach

Benefits

- Allows legitimate functionality to run normally
- Deserialization exploits fail to abuse and compromise the system
- Deserialization payloads cannot bypass security controls
- Removes the need to maintain lists (whitelists / blacklists)
- Protection against
  - known and 0-day gadget chains
  - golden gadget chains
  - all deserialization end-points
  - API Abuse
  - Privilege Escalation
  - DoS

New Mitigation Approach

Conclusion

- Java Serialization is insecure by nature
- Very easy to introduce dangerous gadgets inadvertently
- Maintaining lists does not scale
- App Security should not be a responsibility of the user or the developer
- The runtime platform must
  - be **secure-by-default**
  - safeguard the developer’s code from being abused

New Mitigation Approach

Conclusion

Runtime compartmentalization

- Creates a secure environment for untrusted operations such as deserialization

Privilege de-escalation

- Reliably mitigates API Abuse and Privilege Escalation attacks

Runtime virtualization

- Isolates compartments
- Enforces access control
- Protects the security controls
- Tracks tainted data

Thank you

Apostolos Giannakidis

@apgiannakidis