

Automated Discovery of Deserialization Gadget Chains

Ian Haken
DEF CON 26, 2018

NETFLIX

\$ whoami

Senior Security Software Engineer on Netflix's Platform Security team. Netflix is a microservice ecosystem and we build security services and libraries to keep those applications safe:

- [Secrets at Scale](#)
- [Crypto as a service](#)
- [Netflix-wide Authorization](#)

 [@ianhaken](#)

 <https://github.com/JackOfMostTrades>

Deserialization Gadget Chains

- What is a deserialization vulnerability?
- A brief history of deserialization vulnerabilities
- What is a deserialization gadget chain?
- Why focus on gadget chains?
- Building a tool to find gadget chains
- Exploits discovered

What is a Deserialization Vulnerability?

What is a Deserialization Vulnerability?

In object oriented languages (like Java), data is contained in classes and classes contain code.

What is a Deserialization Vulnerability?

In object oriented languages (like Java), data is contained in classes and classes contain code.

Controlling Data Types => Controlling Code!

What is a Deserialization Vulnerability?

In object oriented languages (like Java), data is contained in classes and classes contain code.

Controlling Data Types => Controlling Code!

```
@POST
public String renderUser(
    HttpServletRequest request) {
    ObjectInputStream ois =
        new ObjectInputStream(
            request.getInputStream());
    User user = (User) ois.readObject();
    return user.render();
}
```

What is a Deserialization Vulnerability?

In object oriented languages (like Java), data is contained in classes and classes contain code.

Controlling Data Types => Controlling Code!

```
@POST
public String renderUser(
    HttpServletRequest request) {
    ObjectInputStream ois =
        new ObjectInputStream(
            request.getInputStream());
    User user = (User) ois.readObject();
    return user.render();
}
```

```
public class User {
    private String name;
    public String render() {
        return name;
    }
}
```


What is a Deserialization Vulnerability?

In object oriented languages (like Java), data is contained in classes and classes contain code.

Controlling Data Types => Controlling Code!

```
@POST
public String renderUser(
    HttpServletRequest request) {
    ObjectInputStream ois =
        new ObjectInputStream(
            request.getInputStream());
    User user = (User) ois.readObject();
    return user.render();
}
```

```
public class User {
    private String name;
    public String render() {
        return name;
    }
}
```

```
public class ThumbnailUser
    extends User {
    private File thumbnail;
    public String render() {
        return Files.read(thumbnail);
    }
}
```

Deserialization? That's so 2016...

1 <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

2 <https://frohoff.github.io/appseccali-marshalling-pickles/>

3 <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

4 <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

5 <https://appsecus2018.sched.com/event/F04J>

Deserialization? That's so 2016...

- 2006: Pentesting J2EE, Black Hat 2006, Marc Schoenefeld¹

¹ <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

² <https://frohoff.github.io/appseccali-marshalling-pickles/>

³ <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

⁴ <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁵ <https://appsecus2018.sched.com/event/F04J>

Deserialization? That's so 2016...

- 2006: Pentesting J2EE, Black Hat 2006, Marc Schönefeld¹
- 2015: Marshalling Pickles, AppSecCali 2015, Frohoff and Lawrence²

¹ <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

² <https://frohoff.github.io/appseccali-marshalling-pickles/>

³ <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

⁴ <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁵ <https://appsecus2018.sched.com/event/F04J>

Deserialization? That's so 2016...

- 2006: Pentesting J2EE, Black Hat 2006, Marc Schönefeld¹
- 2015: Marshalling Pickles, AppSecCali 2015, Frohoff and Lawrence²
- 2016: Defending against Java Deserialization Vulnerabilities, Bay Area OWASP Meetup, September 2016, Luca Carettoni³

¹ <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

² <https://frohoff.github.io/appseccali-marshalling-pickles/>

³ <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

⁴ <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁵ <https://appsecus2018.sched.com/event/F04J>

Deserialization? That's so 2016...

- 2006: Pentesting J2EE, Black Hat 2006, Marc Schönefeld¹
- 2015: Marshalling Pickles, AppSecCali 2015, Frohoff and Lawrence²
- 2016: Defending against Java Deserialization Vulnerabilities, Bay Area OWASP Meetup, September 2016, Luca Carettoni³
- 2017: Friday the 13th: JSON Attacks, Black Hat 2017, Muñoz and Mirosh⁴

¹ <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

² <https://frohoff.github.io/appseccali-marshalling-pickles/>

³ <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

⁴ <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁵ <https://appsecus2018.sched.com/event/F04J>

Deserialization? That's so 2016...

- 2006: Pentesting J2EE, Black Hat 2006, Marc Schönefeld¹
- 2015: Marshalling Pickles, AppSecCali 2015, Frohoff and Lawrence²
- 2016: Defending against Java Deserialization Vulnerabilities, Bay Area OWASP Meetup, September 2016, Luca Carettoni³
- 2017: Friday the 13th: JSON Attacks, Black Hat 2017, Muñoz and Mirosh⁴
- 2018: Deserialization: what, how and why [not], AppSec USA, October 2018, Alexei Kojenov⁵

¹ <http://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Schoenefeld-up.pdf>

² <https://frohoff.github.io/appseccali-marshalling-pickles/>

³ <https://www.slideshare.net/ikkisoft/defending-against-java-deserialization-vulnerabilities>

⁴ <https://www.blackhat.com/docs/us-17/thursday.us-17-Munoz-Friday-The-13th-Json-Attacks.pdf>

⁵ <https://appsecus2018.sched.com/event/F04J>

Why are Deserialization Vulnerabilities so Bad?

Magic methods get executed automatically by the deserializer, even before deserialization finishes!

Why are Deserialization Vulnerabilities so Bad?

Magic methods get executed automatically by the deserializer, even before deserialization finishes!

```
@POST
public String renderUser(
    HttpServletRequest request) {
    ObjectInputStream ois =
        new ObjectInputStream(
            request.getInputStream());
    User user = (User) ois.readObject();
    return user.render();
}
```

Why are Deserialization Vulnerabilities so Bad?

Magic methods get executed automatically by the deserializer, even before deserialization finishes!

```
@POST
public String renderUser(
    HttpServletRequest request) {
    ObjectInputStream ois =
        new ObjectInputStream(
            request.getInputStream());
    User user = (User) ois.readObject();
    return user.render();
}
```

```
public class EvilClass {
    public void readObject(
        ObjectInputStream ois) {
        Runtime.exec(ois.readObject());
    }
}
```

Magic methods?


- `readObject()` and `readResolve()` are the main ones...
 - But don't forget about `finalize()`!
- Many serializable JDK classes implement these magic methods and call other methods, so there's a lot of additional "known entrypoints."
 - `HashMap`
 - `Object.hashCode()`
 - `Object.equals()`
 - `PriorityQueue`
 - `Comparator.compare()`
 - `Comparable.compareTo()`

Magic Methods to Gadget Chains

```
public class HashMap<K,V> implements Map<K,V> {  
    private void readObject(ObjectInputStream s) {  
        int mappings = s.readInt();  
        for (int i = 0; i < mappings; i++) {  
            K key = (K) s.readObject();  
            V value = (V) s.readObject();  
            putVal(key.hashCode(), key, value);  
        }  
    }  
}
```

Magic Methods to Gadget Chains

```
public class HashMap<K,V> implements Map<K,V> {  
    private void readObject(ObjectInputStream s) {  
        int mappings = s.readInt();  
        for (int i = 0; i < mappings; i++) {  
            K key = (K) s.readObject();  
            V value = (V) s.readObject();  
            putVal(key.hashCode(), key, value);  
        }  
    }  
}
```



```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __clojureFnMap;  
    public int hashCode() {  
        IFn f = __clojureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

Magic Methods to Gadget Chains

```
public class HashMap<K,V> implements Map<K,V> {  
    private void readObject(ObjectInputStream s) {  
        int mappings = s.readInt();  
        for (int i = 0; i < mappings; i++) {  
            K key = (K) s.readObject();  
            V value = (V) s.readObject();  
            putVal(key.hashCode(), key, value);  
        }  
    }  
}
```

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) f.invoke(this);  
    }  
}
```

Magic Methods to Gadget Chains

```
public class HashMap<K,V> implements Map<K,V> {  
    private void readObject(ObjectInputStream s) {  
        int mappings = s.readInt();  
        for (int i = 0; i < mappings; i++) {  
            K key = (K) s.readObject();  
            V value = (V) s.readObject();  
            putVal(key.hashCode(), key, value);  
        }  
    }  
}
```

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) f.invoke(this);  
    }  
}
```

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

Magic Methods to Gadget Chains

```
public class HashMap<K,V> implements Map<K,V> {  
    private void readObject(ObjectInputStream s) {  
        int mappings = s.readInt();  
        for (int i = 0; i < mappings; i++) {  
            K key = (K) s.readObject();  
            V value = (V) s.readObject();  
            putVal(key.hashCode(), key, value);  
        }  
    }  
}
```

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) f.invoke(this);  
    }  
}
```

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

```
public class FnEval implements IFn {  
    public Object invoke(Object arg) {  
        return Runtime.exec(arg);  
    }  
}
```



Example Payload

```
{
  "@class": "java.util.HashMap"
  "members": [
    2,
    {
      "@class": "AbstractTableModel$fff19274a"
      __clojureFnMap: {
        hashCode: {
          "@class": "FnCompose"
          f2: { "@class": "FnConstant", value: "/usr/bin/calc" },
          f1: { "@class": "FnEval" }
        }
      }
    },
    "val"
  ]
}
```

What gadget chains are in your application has nothing to do with code your application is calling!

Possible gadget chains are influenced by the cumulative collection of all transitive dependencies for your application

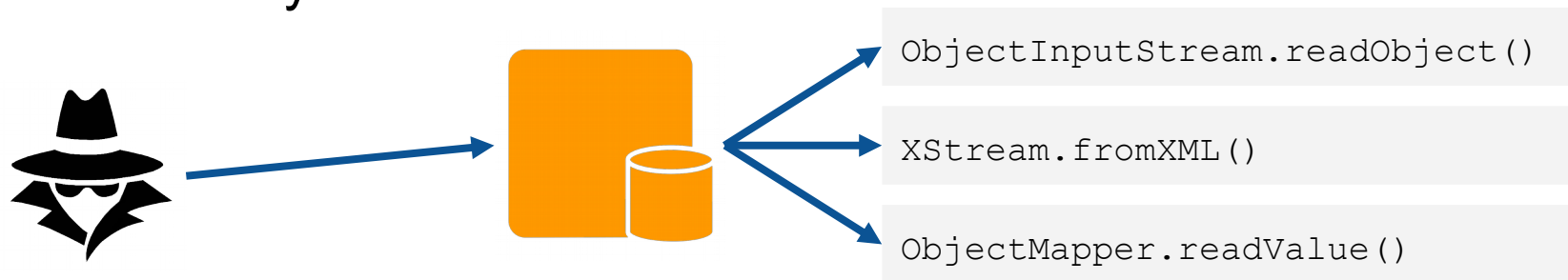
What (Java) Libraries are Vulnerable?

- JDK (ObjectInputStream)
 - XStream (XML, JSON)
 - Jackson (JSON)
 - Genson (JSON)
 - JSON-IO (JSON)
 - FlexSON (JSON)
- 
- Spend some time reading [Muñoz and Mirosh!](#)

Libraries have different behavior about what classes they'll deserialize and what “magic methods” can automatically be invoked. Keep this in mind for later...

Finding Vulnerabilities

- Finding potential vulnerabilities is similar to finding many application security issues:
 - Does untrusted input (e.g. a request's body or input stream) flow into one of the sinks for this vulnerability type?
 - Existing static and dynamic tools are pretty good at discovery



Remediation Options

Remediation Options

- Why not use a better serialization strategy?

Remediation Options

- Why not use a better serialization strategy?
 - “It’s 2016, there are better options.” -Luca Carettoni

Remediation Options

- Why not use a better serialization strategy?
 - “It’s 2016, there are better options.” -Luca Carettoni



Remediation Options

- Why not use a better serialization strategy?
 - “It’s 2016, there are better options.” -Luca Carettoni



- Is it worth the effort to remediate? How should we prioritize remediation?

Is my deserialization vulnerability exploitable?

Finding Exploits

- Known exploits in a few projects:
 - ysoserial: Usually limited to chains in particular libraries and focused on JDK ObjectInputStream
 - marshalsec: Wider breadth of exploits for alternative deserialization libraries
- But what about...
 - The specific *combination* of libraries on my classpath?
 - The non-standard deserialization library that I'm using?

Existing Gadget Chain Tools

- ysoserial¹
 - Collection of known gadget chains and exploits
- joogole²
 - Programatically query about types/methods on the classpath
- Java Deserialization Scanner³
 - Burp Suite plugin; uses known payloads (ysoserial) to discover and exploit vulns.
- marshalsec⁴
 - Deserialization payload generator for numerous libraries and gadget chains
- NCC Group Burp Plugin⁵
 - “Mainly based on the work of Muñoz and Mirosh’s Friday the 13th: JSON Attacks”

¹ <https://github.com/frohoff/ysoserial>

² <https://github.com/Contrast-Security-OSS/joogole>

³ <https://techblog.mediaservice.net/2017/05/reliable-discovery-and-exploitation-of-java-deserialization-vulnerabilities/>

⁴ <https://github.com/mbechler/marshalsec>

⁵ <https://www.nccgroup.trust/us/about-us/newsroom-and-events/blog/2018/june/finding-deserialisation-issues-has-never-been-easier-freddy-the-serialisation-killer/>

Building a New Tool to Find Gadget Chains

What we really want is a tool to evaluate risk.
How important is it to remediate this vulnerability?

- Is a given deserialization vulnerability exploitable?
- What exploits are possible? RCE, SSRF, DoS?
- It doesn't need to be perfect; a reasonable overestimation of risk is useful in this context.
- Note: we don't actually have to actually generate payloads.

Requirements

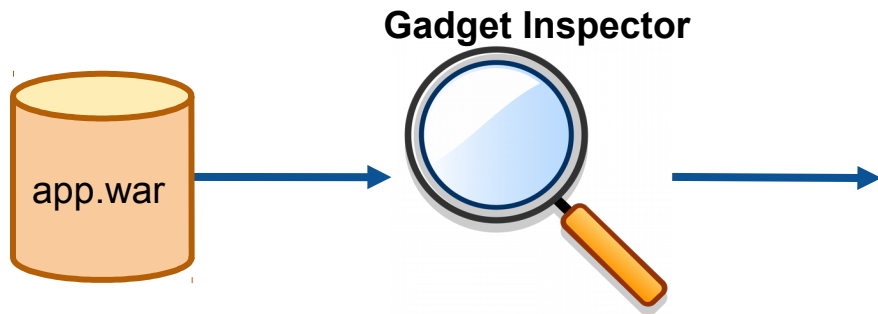
- Not looking for vulnerabilities; assume we only use this tool if we find a vulnerability
- It needs to look at the entire classpath of the application
- It should err on the side of false positives rather than false negatives
- It should operate on bytecode; we usually have the entire classpath packaged as a war and may not have sources (especially if we're including proprietary, third-party libraries)
 - Plus it may include libraries written in Groovy, Scala, Clojure, ...

Gadget Inspector

A Java bytecode analysis tool for finding gadget chains.

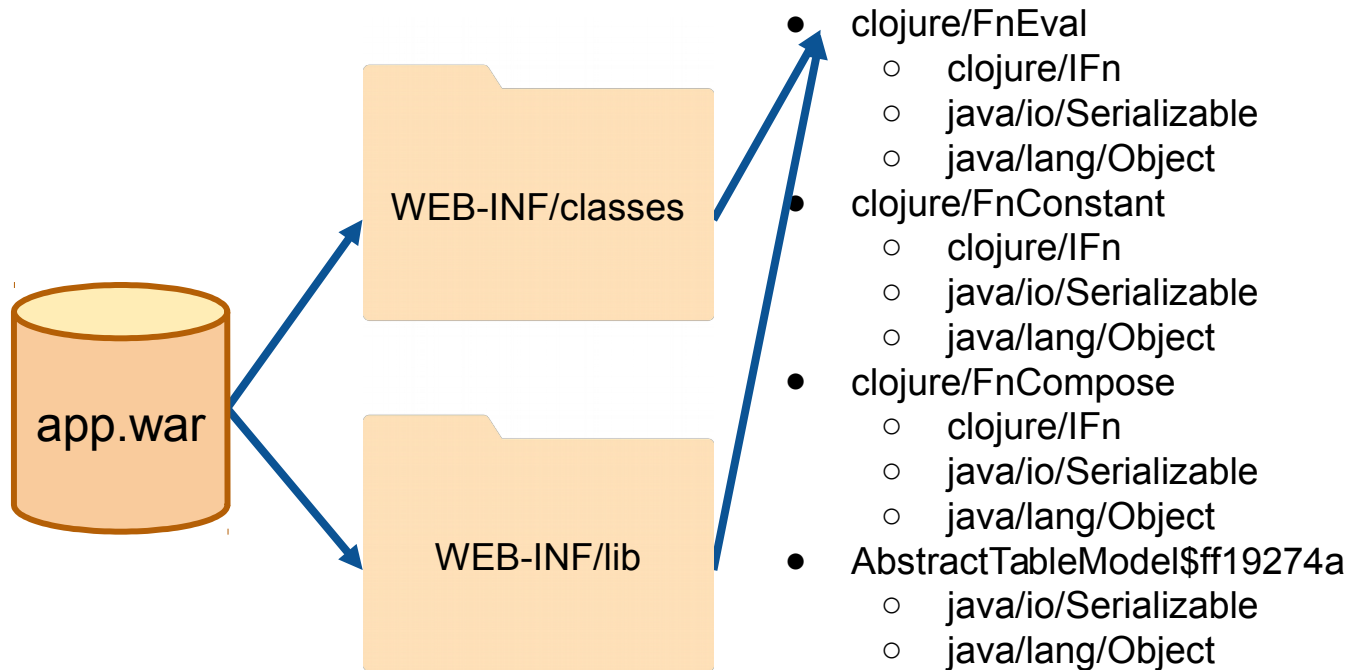
Gadget Inspector

- Operates on any given classpath, i.e. a particular library or an entire war
- Reports discovered gadget chains as a sequence of method invocations
- Performs some simplistic symbolic execution to understand possible dataflow from method arguments to subsequent method invocations
- Makes a lot of simplifying assumptions that make code analysis easy



- `CertificateRevokedException.readObject()`
- `Collections$CheckedMap.put()`
- `TreeMap.put()`
- `scala/math/Ordering$$anon$5.compare()`
- `scala/PartialFunction$OrElse.apply()`
- ...

Step 1: Enumerate class/method hierarchy



Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

- FnConstant.invoke() -> 0

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

- FnConstant.invoke() -> 0
- 

```
public class FnDefault {  
    private FnConstant f;  
    public Object invoke(Object arg) {  
        return arg != null ? arg : f.invoke(arg);  
    }  
}
```

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

- FnConstant.invoke() -> 0
- 

```
public class FnDefault {  
    private FnConstant f;  
    public Object invoke(Object arg) {  
        return arg != null ? arg : f.invoke(arg);  
    }  
}
```

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Assumption #2: All branch conditions are satisfiable

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

- FnConstant.invoke() -> 0
- FnDefault.invoke() -> 1
- FnDefault.invoke() -> 0

```
public class FnDefault {  
    private FnConstant f;  
    public Object invoke(Object arg) {  
        return arg != null ? arg : f.invoke(arg);  
    }  
}
```

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Assumption #2: All branch conditions are satisfiable

Step 2: Discover “Passthrough” Dataflow

```
public class FnConstant implements IFn {  
    private Object value;  
    public Object invoke(Object arg) {  
        return value;  
    }  
}
```

```
public class FnDefault {  
    private FnConstant f;  
    public Object invoke(Object arg) {  
        return arg != null ? arg : f.invoke(arg);  
    }  
}
```

- FnConstant.invoke() -> 0
- FnDefault.invoke() -> 1
- FnDefault.invoke() -> 0

Assumption #1: All members of a “tainted” object are also tainted (and recursively, etc)

Assumption #2: All branch conditions are satisfiable

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$fff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```


Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

Step 3: Enumerate “Passthrough” Callgraph

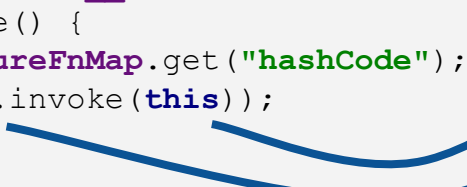
```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```



AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

Step 3: Enumerate “Passthrough” Callgraph

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

```
public class FnCompose implements IFn {  
    private IFn f1, f2;  
    public Object invoke(Object arg) {  
        return f2.invoke(f1.invoke(arg));  
    }  
}
```

FnCompose.invoke()

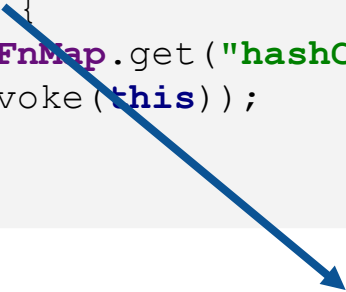
- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

Step 4: Enumerate Sources Using Known Tricks

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __clojureFnMap;  
    public int hashCode() {  
        IFn f = __clojureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```


Step 4: Enumerate Sources Using Known Tricks

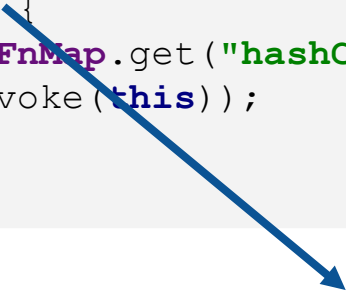
```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```



AbstractTableModel\$ff19274a.hashCode() @ 0

Step 4: Enumerate Sources Using Known Tricks

```
public class AbstractTableModel$ff19274a {  
    private IPersistentMap __closureFnMap;  
    public int hashCode() {  
        IFn f = __closureFnMap.get("hashCode");  
        return (int) (f.invoke(this));  
    }  
}
```



AbstractTableModel\$ff19274a.hashCode() @ 0

Limitation #1: Relies on known tricks. Some tricks (e.g. HashMap -> hashCode) we could derive anyway, but others (like DynamicProxy) we could not.

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Assumption #3: Any method implementation can be jumped to (as long as its class is “serializable”)

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0
- FnCompose.invoke() @ 0

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Assumption #3: Any method implementation can be jumped to (as long as its class is “serializable”)

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0
- FnCompose.invoke() @ 0
- FnEval @ 1

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Assumption #3: Any method implementation can be jumped to (as long as its class is “serializable”)

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0
- FnCompose.invoke() @ 0
- FnEval @ 1
- Runtime.exec() @ 1

Call Graph

AbstractTableModel\$ff19274a.hashCode()

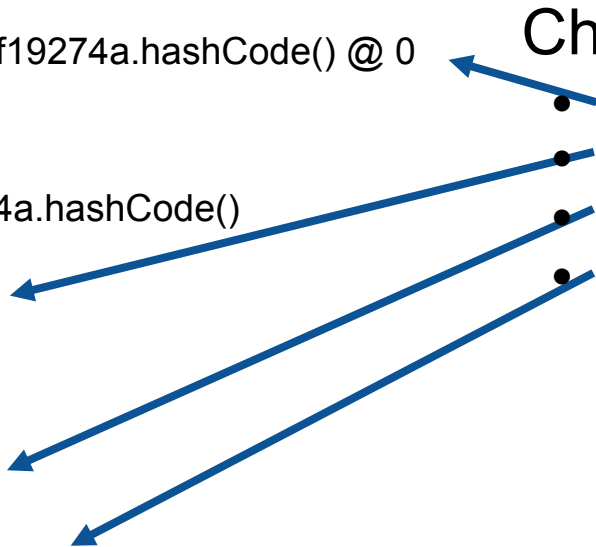
- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1



Assumption #3: Any method implementation can be jumped to (as long as its class is “serializable”)

Step 5: BFS on Call Graph for Chains

Sources

- AbstractTableModel\$ff19274a.hashCode() @ 0

Chain

- AbstractTableModel...hashCode() @ 0
- FnCompose.invoke() @ 0
- FnEval @ 1
- Runtime.exec() @ 1

Call Graph

AbstractTableModel\$ff19274a.hashCode()

- 0 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0

FnCompose.invoke()

- 1 -> IFn.invoke() @ 1
- 0 -> IFn.invoke() @ 0
- 0 -> IFn.invoke() @ 1

FnEval.invoke()

- 1 -> Runtime.exec() @ 1

Assumption #3: Any method implementation can be jumped to (as long as its class is “serializable”)

Limitation #2: Chain discovery relies on a list of known “interesting” sinks

Deserialization Library Flexibility

Gadget Inspector supports some customization on the analysis

- What is considered “serializable”?
 - For JRE deserialization, anything implementing `java.lang.Serializable`
 - For XStream, it depends on the converters that are enabled
 - And when using custom converters, it gets even more subtle
 - For Jackson, any class with a no-arg constructor
- What are the deserialization sources (i.e. magic methods)?
 - For Jackson we only start in constructors
- What method implementations should we consider?
 - For JRE deserialization, all implementations in a serializable class
 - For Jackson, depends on annotations and configuration

OSS Library Results

Results: OSS Library Scans

Ran Gadget Inspector against the 100 most popular java libraries (according to mvnrepository.com and javalibs.com) looking for exploits against standard Java deserialization

- It did rediscover several known gadget chains
- Not that many libraries actually have classes implementing `java.io.Serializable`...
 - But there were some interesting new findings!
- Had a handful of false positives but not as many as you'd expect
 - Mostly because reflection is hard to reason about

Results: Old Gadget Chains

commons-collections » commons-collections

38th most popular maven dependency

1.corba.se.spi.orbutil.proxy.CompositeInvocationHandlerImpl
invoke(Object, Method, Object[]) (0)
2. org.apache.commons.collections.map.LazyMap.get(Object) (0)
3. org.apache.commons.collections.functors.InvokerTransformer
transform(Object) (0)
4. java.lang.reflect.Method.invoke(Object, Object[]) (0)

<https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/CommonsCollections1.java>

New Gadget Chains: Clojure

org.clojure » clojure

6th most popular maven dependency

1. clojure.inspector.proxy\$javax.swing.table.AbstractTableModel\$ff19274a.hashCode() (0)
2. clojure.main\$load_script.invoke(Object) (1)
3. clojure.main\$load_script.invokeStatic(Object) (0)
4. clojure.lang.Compiler.loadFile(String) (0)
5. FileInputStream.<init>(String) (1)

Tweaked this result to invoke clojure.main\$eval_opt instead of clojure.main\$load_script to invoke arbitrary code.

Reported to clojure-dev July 2017, affecting 1.8.0 and all earlier versions.

Serialization of **AbstractTableModel\$ff19274a** disabled in 1.9.0 release (Dec, 2017).

<https://github.com/frohoff/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure.java>

New Gadget Chains: Scala

org.scala-lang » scala-library

3rd most popular maven dependency

1. `scala.math.Ordering$$anon$5.compare(Object, Object) (0)`
2. `scala.PartialFunction$OrElse.apply(Object) (0)`
3. `scala.sys.process.processInternal$$anonfun$onInterrupt$1
 applyOrElse(Object, scala.Function1) (0)`
4. `scala.sys.process.ProcessBuilderImpl$FileOutput$$anonfun$$lessinit$greater$3.apply() (0)`
5. `java.io.FileOutputStream.<init>(File, boolean) (1)`

Allows you to write/overwrite a file with 0 bytes.

Possible DoS? Zero-out a blacklist?

<https://github.com/JackOfMostTrades/ysoserial/blob/master/src/main/java/ysoserial/payloads/Scala.java>

New Gadget Chains: Scala

1. `scala.math.Ordering$$anon$5.compare(Object, Object) (0)`
2. `scala.PartialFunction$OrElse.apply(Object) (0)`
3. `scala.sys.process.processInternal$$anonfun$onIOInterrupt$1
 applyOrElse(Object, scala.Function1) (0)`
4. `scala.sys.process.ProcessBuilderImpl$URLInput$$anonfun$
 $lessinit$greater$1.apply() (0)`
5. `java.net.URL.openStream() (0)`

SSRF: Causes application to perform a GET on an arbitrary URL.

<https://github.com/JackOfMostTrades/ysoserial/blob/master/src/main/java/ysoserial/payloads/Scala.java>

New Gadget Chains: Clojure2

In rerunning Gadget Inspector on the latest release (1.10.0-alpha4) in preparation for this presentation, a different entry point was discovered:

1. `clojure.lang.ASeq.hashCode()` (0)
2. `clojure.lang.Iterate.first()` (0)
3. `clojure.main$load_script.invoke(Object)` (1)
4. `clojure.main$load_script.invokeStatic(Object)` (0)
5. `clojure.lang.Compiler.loadFile(String)` (0)
6. `FileInputStream.<init>(String)` (1)

Confirmed the same tweak to `clojure.main$eval_opt` works for arbitrary code execution. Affects all releases since 1.8.0.

<https://github.com/JackOfMostTrades/ysoserial/blob/master/src/main/java/ysoserial/payloads/Clojure2.java>

Netflix App Results

Results: Netflix Internal Webapp 1

Potentially dangerous use of Jackson deserialization:

```
public void doSomething(String body, String queryParams) {  
    Object requestObject = objectMapper.readValue(  
        body, Class.forName(queryParams));  
}
```

- Can only deserialize classes with no-arg constructors
- The only entry points are no-arg constructors

But still, the app has ~200MB classpath. So maybe there's something there...

Results: Netflix Internal Webapp 1

Potentially dangerous use of Jackson deserialization:

```
public void doSomething(String body, String queryParams) {  
    Object requestObject = objectMapper.readValue(  
        body, Class.forName(queryParams));  
}
```

- Can only deserialize classes with no-arg constructors
- The only entry points are no-arg constructors

But still, the app has ~200MB classpath. So maybe there's something there...

Result: A few false positives and nothing very interesting.

Take-away: Remediation is a low priority

Results: Netflix Internal Webapp 2

Used a non-standard deserialization library, subject to some unique constraints

- Invokes readResolve() but not readObject()
- Serialized objects do *not* need to implement Serializable
- Member fields of serialized objects cannot have a \$ in the name.
 - Non-static inner classes always have an implicit \$outer member name.
- No serialization support for arrays or generic maps
- No null member values

Results: Netflix Internal Webapp 2

1. `com.thoughtworks.xstream.mapper.AbstractAttributeAliasingMapper.readResolve()` (0)
2. `org.apache.commons.configuration.ConfigurationMap$ConfigurationSet.iterator()` (0)
3. `...configuration.ConfigurationMap$ConfigurationSet$ConfigurationSetIterator.<init>()` (0)
4. `org.apache.commons.configuration.CompositeConfiguration.getKeys()` (0)
5. `clojure.lang.APersistentMap$KeySeq.iterator()` (0)
6. `com.netflix.internal.utils.collections.IteratorWrapper$CallableWrapper.iterator()` (0)
7. `java.util.concurrent.Executors$RunnableAdapter.call()` (0)
8. `org.apache.commons.exec.StreamPumper.run()` (0)
9. `org.eclipse.core.internal.localstore.SafeFileOutputStream.close()` (0)
10. `org.eclipse.core.internal.localstore.SafeFileOutputStream.commit()` (0)
11. `org.eclipse.core.internal.localstore.SafeFileOutputStream.copy(File, File)` (2)
12. `java.io.FileOutputStream.<init>(File)` (1)

Results: Netflix Internal Webapp 2

1. `com.thoughtworks.xstream.mapper.AbstractAttributeAliasingMapper.readResolve() (0)`
2. `org.apache.commons.configuration.ConfigurationMap$ConfigurationSet.iterator() (0)`
3. `...configuration.ConfigurationMap$ConfigurationSet$ConfigurationSetIterator.<init>() (0)`
4. `org.apache.commons.configuration.CompositeConfiguration.getKeys() (0)`
5. `clojure.lang.APersistentMap$KeySeq.iterator() (0)`
6. `com.netflix.internal.utils.collections.IteratorWrapper$CallableWrapper.iterator() (0)`
7. `java.util.concurrent.Executors$RunnableAdapter.call() (0)`
8. `org.apache.commons.exec.StreamPumper.run() (0)`
9. `org.eclipse.core.internal.localstore.SafeFileOutputStream.close() (0)`
10. `org.eclipse.core.internal.localstore.SafeFileOutputStream.commit() (0)`
11. `org.eclipse.core.internal.localstore.SafeFileOutputStream.copy(File, File) (2)`
12. `java.io.FileOutputStream.<init>(File) (1)`

`com.thoughtworks.xstream:xstream`

`commons-configuration:commons-configuration`

`org.clojure:clojure`

`netflix:netflix-utils`

`JRE`

`org.apache.commons:commons-exec`

`org.aspectj:aspectjtools`

Results: Netflix Internal Webapp 2

1. com.thoughtworks.xstream.mapper.AbstractAttributeAliasingMapper.readResolve() (0)
2. org.apache.commons.configuration.ConfigurationMap\$ConfigurationSet.iterator() (0)
3. ...configuration.ConfigurationMap\$ConfigurationSet\$ConfigurationSetIterator.<init>() (0)
4. org.apache.commons.configuration.CompositeConfiguration.getKeys() (0)
5. clojure.lang.APersistentMap\$KeySeq.iterator() (0)
6. com.netflix.internal.utils.collections.IteratorWrapper\$CallableWrapper.iterator() (0)
7. java.util.concurrent.Executors\$RunnableAdapter.call() (0)
8. org.apache.commons.exec.StreamPumper.run() (0)
9. org.eclipse.core.internal.localstore.SafeFileOutputStream.close() (0)
10. org.eclipse.core.internal.localstore.SafeFileOutputStream.commit() (0)
11. org.eclipse.core.internal.localstore.SafeFileOutputStream.copy(File, File) (2)
12. java.io.FileOutputStream.<init>(File) (1)

Results: Netflix Internal Webapp 2

1. com.thoughtworks.xstream.mapper.AbstractAttributeAliasingMapper.readResolve() (0)
2. org.apache.commons.configuration.ConfigurationMap\$ConfigurationSet.iterator() (0)
3. ...configuration.ConfigurationMap\$ConfigurationSet\$ConfigurationSetIterator.<init>() (0)
4. org.apache.commons.configuration.CompositeConfiguration.getKeys() (0)
5. clojure.lang.APersistentMap\$KeySeq.iterator() (0)
6. com.netflix.internal.utils.collections.IteratorWrapper\$CallableWrapper.iterator() (0)
7. java.util.concurrent.Executors\$RunnableAdapter.call() (0)
8. org.apache.commons.exec.StreamPumper.run() (0)
 - is = java.io.StringBufferInputStream
 - buffer = <% String cmd="calc.exe"; ... %>
 - os = org.eclipse.core.internal.localstore.SafeChunkyOutputStream
 - isOpen = false
 - filePath = /webappdir/foo.jsp

Room for Improvement

- Reflection

- Most reflection calls are being treated as interesting, leading to FPs
 - E.g. you can control the class but not the method name, or vice-versa
- Blind spots for call graph enumeration
 - `foo.getClass().getMethod("bar").invoke(...)`

- Assumptions

- Even minor improvements would allow Gadget Inspector to make better decisions around condition satisfiability or virtual method call resolution, leading to fewer FPs.

- Limitations

- Entry points are enumerated using “known tricks.” Original research can still help us find lots of other clever ways to construct gadget chains.
- Sinks with “interesting behavior” are hard-coded. Lots of room to discover and add sinks.

Final Thoughts

- Automatic discovery for gadget chains is new territory
 - Gadget Inspector is a functional prototype; room for lots of improvement!
 - Gadget Inspector written for Java but techniques apply to other languages
- Gadget Inspector is open source
 - Fork it, submit PRs, or just check it out for more details about how it works
 - <https://github.com/JackOfMostTrades/gadgetinspector>
- Deserialization vulnerabilities aren't going away yet
 - Exploits can and will be more complex as time goes on
 - Better tools will help us understand the risk of vulnerabilities



[@ianhaken](https://twitter.com/ianhaken)



<https://github.com/JackOfMostTrades>

NETFLIX