

# INF889A

## Analyse de programmes pour la sécurité logicielle

GadgetInspector

---

Philippe Grégoire

2024-04-12

Université du Québec à Montréal

- L'outil *GadgetInspector*
- Développé par Ian Haken
- Publié en 2018 à *Black Hat USA*<sup>1</sup>
- Inspecte les classes Java disponibles pour des chaînes de gadgets de désérialisation avec des effets de sécurité intéressants
- Permet de découvrir si une désérialisation non-sécuritaire est exploitable

---

<sup>1</sup> Ian Haken - Automated Discovery of Deserialization Gadget Chains

- Traite directement de mon sujet de mémoire
- Fais partie de l'étude de l'état de l'art pour ma recherche
- Un des rares outils disponibles<sup>2</sup> sur ce sujet

---

<sup>2</sup>aussi [frohoff/inspector-gadget](#)

# Le programme

- Une brève introduction à la sérialisation
- Les vulnérabilités de désérialisation Java
- Une exploration de GadgetInspector
- Une démonstration
- La conclusion

# La sérialisation

---

*[...] the process of translating a data structure or object state into a format that can be stored or transmitted, and reconstructed later.*

*Wikipedia, Serialization*

- Des formats génériques: XML, YAML, TOML, etc.
- Et des formats natifs, par langage

## Quelques exemples

```
// ECMAScript, {"a": 1}  
JSON.stringify({a: 1})  
// PHP, a:1:{s:1:"a";i:1;}  
serialize(array('a'=>1));  
// Ruby, \x04\b{\x06I"\x06a\x06:\x06ETi\x06  
Marshal.dump({'a'=>1})  
// Python, b'\x80\x04\x95\n\x00\x00\x00...'   
pickle.dumps({'a': 1})
```

- Catégorisée **CWE-502: Deserialization of Untrusted Data**
- Lorsqu'il est possible de désérialiser un objet arbitraire...
- et modifier l'état ou le comportement du programme
- Top 8, selon OWASP Top Ten 2017



# **Les vulnérabilités de désérialisation Java**

---

- Les interfaces `Serializable` et `Externalizable` permettent de sérialiser
- Par défaut, (dé)sérialisent les attributs d'instance
- Certaines méthodes permettent de modifier ces comportements
- e.g. `readObject`, `writeObject`, `readExternal`, etc.

## Un exemple

Exécution d'une commande système à partir d'une désérialisation.

- `readObject` est une source
- `Runtime.exec` est un puit

```
class Foo {  
    public void readObject(java.io.ObjectInputStream in)  
    {  
        Runtime.getRuntime().exec(in.readUTF());  
    }  
}
```

## Un exemple

Exécution d'une commande système à partir d'une désérialisation.

```
class Foo {  
    private String cmd;  
    public void runCmd() {  
        System.getRuntime().exec(this.cmd);  
    }  
}
```

- Un attaquant doit envoyer une instance de Foo sérialisée
- Le programme doit appeler runCmd() sur l'objet désérialisé
- En théorie, si runCmd n'est jamais appelé, tout va bien.

En 2015, Frohoff et Lawrence ont démontré<sup>3</sup> qu'il est possible de réutiliser du code Java existant (des gadgets) pour, dans le pire cas, exécuter du code Java arbitraire ou des commandes systèmes.

Depuis que la technique a été partagée, un ensemble<sup>4</sup> de vulnérabilités ont été découvertes, et de recherches effectués pour tenter de faciliter la découverte de gadgets et la construction de chaines utiles.

---

<sup>3</sup>Frohoff et Lawrence - Marshalling Pickles

<sup>4</sup>Aleksei Tiurin - Java Deserialization Cheat Sheet

L'exploitation de désérialisation non-sécuritaire requiert généralement des objets sérialisés complexes

- La chaine URLDNS cause une requête DNS à sa désérialisation
  - Il s'agit d'un HashMap contenant un URL spécialement conçu
  - La requête DNS peut être observée par l'attaquant pour confirmer la faiblesse

```
HashMap.readObject()    # reconstruit son état
HashMap.putVal()        # insertion d'une valeur
HashMap.hash()          # test d'égalité
URL.hashCode()          # requête DNS
```

Notez que le code de `readObject` n'appelle jamais `URL.hashCode()` directement.

## Quelques instances connues

- *Apache Commons Collections*
  - CVE-2015-7501, CVE-2015-6420
- *Oracle WebLogic*
  - CVE-2015-4852, ..., CVE-2023-21931
- *Red Hat JBoss*
  - CVE-2015-7501
- *IBM WebSphere, Apache Tomcat, Jackson, SnakeYAML, Jenkins*
- CORBA, RMI, IIOP, JMX, JNDI, T3
- Bien plus encore...

- On identifie les classes qui implémentent `Serializable` ou `Externalizable`
- On détermine si elles implémentent `readObject` (et similaires)
- On étudie la méthode pour un comportement intéressant
  - Probablement en suivant la chaîne d'appels



# Les obstacles à l'exploitation

- On doit trouver une source de désérialisation
- On doit trouver comment atteindre cette source
  - Il est possible que la source soit inatteignable<sup>5</sup>
- Généralement, on rapporte uniquement les chaines exploitables
- Mais, si la chaine est dans une librairie, la chaine suffit...
  - Même si certains développeurs se ferment les yeux<sup>67</sup>

---

<sup>5</sup>Inversement, on peut avoir un source sans chaine

<sup>6</sup>snakeyaml - CVE & NIST

<sup>7</sup>snakeyaml - Billion laughs attack

# L'outil GadgetInspector

---

- <https://github.com/JackOfMostTrades/gadgetinspector>
- Écrit en Java, 3124 lignes de code
- Aucun changement depuis 5 ans
- Quelques *issues* et *pull requests* ouverts

# Le programme

- On fournit un ensemble de JARs ou WARs en entrée
- Cinq (5) passes d'analyse statique
- Utilise la librairie Java ASM pour analyser le *bytecode*
- En sortie, gadget-chains.txt avec les chaines détectées

*# toutes les passes*

```
java -jar gadget-inspector-all.jar jar...
```

*# des passes spécifiques sur des WAR seulement*

```
java -cp gadget-inspector-all.jar \  
    gadgetinspector.$Passe war...
```

- Énumère les classes et méthodes vers `classes.dat` et `methods.dat`
- Construit l'arbre d'héritage vers `inheritanceMap.dat`
- Pour les classes:
  - son nom;
  - sa super-classe;
  - ses interfaces;
  - s'il s'agit d'une interface;
  - ses attributs.
- Pour les méthodes:
  - son nom;
  - son descripteur;
  - si elle est statique;
  - sa classe de définition.

---

<sup>8</sup> `MethodDiscovery.java`

<sup>9</sup> `ClassReference.java`

<sup>10</sup> `MethodReference.java`

## La passe 2 - PassthroughDiscovery

- **Comment les données circulent-elles?**
- À partir des fichiers produits par la passe précédente
- Produit `passthrough.dat`
- Construit le CFG<sup>11</sup> avec un algorithme DFS
- Ensuite, pour chaque méthode:
  - Identifie les variables d'instances et de pile sérialisables
  - Effectue une analyse de teinte<sup>1213</sup> sur les retour et les paramètres
- Soit les méthodes qui reçoivent ou passent un objet sérialisable
- Pour chaque résultat, enregistre: la classe, la méthode, le descripteur, la position des arguments teintés

---

<sup>11</sup> [PassthroughDiscovery.java#L148](#)

<sup>12</sup> [PassthroughDiscovery.java#L301](#)

<sup>13</sup> [TaintTrackingMethodVisitor.java](#)

```
/*  
 * javax/swing/text/html/HTMLDocument  
 * getBase  
 * ()Ljava/net/URL;  
 * 0,  
 */  
public URL getBase();
```

---

<sup>14</sup>[javax.swing.text.html.HTMLDocument](#)

## La passe 2 - PassthroughDiscovery - keytool/Main<sup>15</sup>

```
/* sun/security/tools/keytool/Main
 * withWeak
 * (Ljava/lang/String;)Ljava/lang/String;
 * 1, */
public final class Main {
    ...
    private String withWeak(String alg) {
        if(DISABLED_CHECK.permits(SIG_PRIMITIVE_SET, alg, null))
        {
            return alg;
        } else {
            return String.format(rb.getString("with.weak"), alg);
        } } }
}
```

---

<sup>15</sup> [openjdk-jdk11 - keytool/Main.java](#)



# Une pause de réflexion

- L'ensemble des classes et méthodes a été identifiées
- Les méthodes qui passent ou reçoivent des objets sérialisables ont été identifiées

Les classes sérialisables n'ont pas été identifiées directement, mais par le biais de l'analyse de teinte.

- Pour certaines super-classes, le programme court-circuite l'analyse; e.g.
  - `java.util.Collection`
  - `java.util.Map`
- On a une liste de gadgets, mais pas de chaine ou de source

## La passe 3 - CallGraphDiscovery

- **Crée un graphe d'appels liés par les paramètres teintés**
- Essentiellement, la passe 2 mais avec les liens
- Les liens sont enregistrés dans `callgraph.dat`

- Utilise la sortie de la première passe
- **Tente d'identifier<sup>16</sup> les sources de désérialisation**
  - Doit implémenté `java.io.Serializable`
  - Sa désérialisation ne doit pas être bloquée par un proxy<sup>17</sup>
- `SourceDiscovery.java` est une façade
  - Une tentative de supporter la désérialisation native, et via `XStream` ou `Jackson`<sup>18</sup>

---

<sup>16</sup>[SimpleSerializableDecider.java](#)

<sup>17</sup>[Serialization Proxy Pattern in Java](#)

<sup>18</sup>Seule la désérialisation native est supportée

Pour chaque classe, est-ce qu'une de ces méthodes existent?

- `void readObject(java.io.ObjectInputStream)`
- `void finalize()`
  - Peut déclencher un comportement intéressant
- `int hashCode()`
  - Appelé dans, e.g., les Map
  - Peut déclencher un comportement intéressant
- `boolean equals(java.lang.Object)`
  - comme `hashCode()`

Est-ce que la classe implémente... ?

- `java.lang.reflect.InvocationHandler` et la méthode `Object invoke(Object, java.lang.reflect.Method, Object[])`?
  - Les chaines utilisant `InvocationHandler` comme proxy
    - `JSON1`, `Jdk7u21`, `Groovy1`, etc.
- `groovy.lang.Closure`, et la méthode `call` ou `doCall`
  - Les chaines utilisant `groovy.MethodClosure` comme gadget
    - `Groovy1`

Notons que ces deux (2) cas gadgets utilisent la réflexivité.

- Pour chacun des candidats trouvés, on enregistre:
  - le nom de la méthode;
  - le nom de sa classe de définition;
  - son descripteur;
  - la position de l'argument qui est considéré teinté.

On a identifié:

- des méthodes “source” et les paramètres teintés;
- des méthodes recevant ou retournant des objets sérialisables;
- des méthodes “teintés” et celles qu’elles invoquent avec ces éléments.

À partir des sources, on peut suivre les paramètres teintés et voir comment ils se propagent dans les méthodes teintées... mais on doit savoir quand arrêter.

### Tente de construire des chaines capables d'atteindre des puits spécifiques

- Exécution de commandes arbitraires
  - `java.lang.Runtime.exec`
  - `java.lang.ProcessBuilder.<init>`
- Déni de service
  - `java.lang.System.exit`
  - `java.lang.System.shutdown`
- Entrée/sortie
  - `java.io.FileInputStream.<init>`
  - `java.io.FileOutputStream.<init>`
- jython, Groovy
- Par réflexivité partielle
  - `java.lang.reflect.Method.invoke` avec le premier argument teinté



- Reprend le graphe d'appels de `callgraph.dat`
- Récupère la liste des sources pour exploration
- On parcourt le graphe selon un algorithme BFS
  - On débute la recherche aux sources
  - Les appels contenant les paramètres teintés sont ajoutés à la liste d'exploration
- Lorsqu'on atteint un puit, on enregistre la chaîne
- Jusqu'à l'épuisement de la liste de candidats

# La démonstration

---

## Les limitations

---

## Les branches inatteignables

L'analyse de GadgetInspector est entièrement statique.

```
private String cmd;
public void readObject(java.io.ObjectInputStream in)
{
    // teinte "cmd"
    in.defaultReadObject();

    // toujours faux
    if ("a".equals("")) {
        // faux positif
        Runtime.getRuntime().exec(cmd);
    }
}
```

L'analyse de teinte s'arrête aux appels non-teintés.

```
private String cmd;  
public void readObject(java.io.ObjectInputStream in)  
{  
    // teinte "cmd"  
    in.defaultReadObject();  
  
    // équivalent à Runtime.getRuntime().exec(cmd);  
    Runtime.class.getMethod("exec", String.class)  
        .invoke(Runtime.getRuntime(), cmd);  
}
```

## D'autres limitations

- Cible Java 8 et inférieure (2014)
- Ne supporte pas `java.io.Externalizable`
- Ne supporte les alternatives à `readObject()`
- Capacité limitée de pistage de teinte
  - Les types primitifs sont ignorées
    - Rend les puits comme `System.exit` inatteignables, en pratique
  - Seules des méthodes spécifiques propages la teinte
    - e.g. `readInt` et `readUTF` sont ignorées
- Ignore les appels via la *Java Native Interface*
- Certains puits bien connus sont absents
- Faux-négatif sur les appels dangereux avec des constantes<sup>19</sup>
- Non-fonctionnel sur du code obfusqué
- Absence de génération d'exploit

---

<sup>19</sup>C'est une question philosophique

- L'ordre d'exécution est respectée
  - `readDefaultObject` en fin de méthode n'a pas d'effet
- Les objets non-teintés provoquent des faux négatifs

- Un mélange de sur- et sous-approximation
  - Génère des faux-positifs **et** des faux-négatifs
- Une approche raisonnable pour des cas simples
- Parvient à identifier certaines chaînes connues et inconnues
- Outil intéressant, mais une applicabilité limitée
- D'autres outils et recherches disponibles en référence



- Ian Haken - GadgetInspector
- Ian Haken - Automated Discovery of Deserialization Gadget Chains
- CWE-502: Deserialization of Untrusted Data
- OWASP - Top Ten 2017
- Chris Frohoff - Inspector-Gadget
- threedr3am - GadgetInspector
- Cao et al. (2023). ODDFUZZ: Discovering Java Deserialization Vulnerabilities via Structure-Aware Directed Greybox Fuzzing
- Aleksei Tiurin - Java Deserialization Cheat Sheet