

INF889A

Analyse de programmes pour la sécurité logicielle

GadgetInspector

Philippe Grégoire

2024-04-26

Université du Québec à Montréal

- Améliorer GadgetInspector pour augmenter le nombre de chaînes de gadgets identifiées

- L'outil fait partie de la revue de littérature de mon mémoire
- Déjà étudié dans le cadre du cours
 - Une relative familiarité avec l'outil
 - Des problèmes déjà identifiés
 - Des pistes de solution déjà en hypothèses

- GadgetInspector
 - 5 passes d'analyse statique
 - Analyse de teinte
 - Chainage des éléments teintés
- Plusieurs limitations relatives et sources et puits

Les objectifs

- Ajouter des sources de désérialisation
 - `java.io.Externalizable`
- Ajouter des sources de teintes
 - D'autres méthodes de `java.io.Serializable`
 - Des méthodes de `java.io.Externalizable`
- Réduire le nombre de faux-négatifs
 - Supporter les types primitifs
- Ajouter des puits d'exploitation
 - Pour la chaîne URLDNS

- L'analyse du flot d'informations
- Les sources de teintes
- Les sources de désérialisation
- Les puits d'exploitation
- La conclusion

L'analyse du flot d'information

La structure de la JVM¹

- En Java, à l'entrée d'une méthode, un *stack frame* est créé
- Il contient:
 - les variables locales et les arguments
 - la pile des opérandes (pour les appels)
 - des méta-données

```
void foo(int arg) {  
    int x = bar(arg);  
    return baz(x);  
}
```

- arg et x
- push arg et push x

¹The Structure of the Java Virtual Machine

Le traçage

- Les attributs d'instance ne sont pas teintés
 - Sauf dans le cas de `defaultReadObject()`
- Les analyses intra- et inter-procédurales sont déficientes
- Perte **cruciale** d'information

```
String x;                                invokevirtual readUTF
                                           putfield x
readObject(in) {                          ...
    x = in.readUTF();                     getfield x
    ...                                   invokevirtual sink
    sink(x);
}
```

La démonstration

Le traçage

- La solution:
 - Lorsque le retour est teinté...
 - les traces associées doivent être transférées

appel d'un objet teinté

invokeinterface readUTF

l'attribut est teinté aussi

putfield x

transfert de la teinte vers la pile

getfield x

on passe la teinte à la méthode appelée

invokeinterface sink

- Une dizaine d'heures passées à étudier comment transférer les teintes
 - À l'intersection de PUTFIELD² et GETFIELD³
 - Aucun résultat concret :(
- Nos tests utiliseront:
 - des variables locales pour les sources de teintes;
 - d'autres trucs, comme on verra.

²Spécification - putfield

³Spécification - getfield

Les sources de teinte

- Certaines méthodes manquantes
 - `String readUTF()`
 - `int read()`
 - `boolean readBoolean()`
 - ...
- Cette absence cause des faux-négatifs

⁴Spécification - `ObjectInputStream`

La démonstration

Les types primitifs teintés

- Les types primitifs ne sont pas tracés
 - Une des causes de faux-négatifs
- Tentative de changer la condition de traçage
 - De Serializable...
 - ...à Serializable et pas un Object
 - Soit: absent de la hiérarchie de classes de la passe 1
 - Faux-négatif d'identification pour les classes manquantes
- En pratique, le *bytecode* des primitifs est différent
- L'analyse doit être adapté en conséquence

```
void readObject(ObjectInputStream in) {  
    int c = in.read();  
    System.exit(c);  
}
```


La démonstration

Les types primitifs teintés

- Une seule exception...
- quand le retour transite directement dans un puit

```
void readObject(ObjectInputStream in) {  
    System.exit(in.read());  
}
```

La démonstration

- Cependant, ça ne passe pas les appels

La démonstration

Les types primitifs teintés

- D'autres sources:
 - `boolean readBoolean()`
 - `byte readByte()`
 - `char readChar()`
- Difficile à tester en l'absence de puits ou de branchements, e.g.
- Support, mais limité
 - Pas de traçage intra-procédurale
 - Pas de traçage inter-procédurale
- Nécessite un effort relativement considérable
 - Ajout des *bytecodes*
 - Adaptation des conditions de traçage

Les sources de désérialisation

- Limité aux classes qui implémentent `java.io.Serializable`
 - `java.io.Externalizable` spécialise `Serializable`
 - On ajoute ses méthodes de désérialisation comme sources
- En pratique, on ajoute:
 - la source de désérialisation `Externalizable.readExternal`
 - la source de teinte `ObjectInput.readObject()`
 - et les autres sources de teinte de `ObjectInput`

La démonstration

Les puits d'exploitation

- La chaîne n'est pas détectée même si elle est native
- Débute à la méthode `hashCode()` de `java.net.URL`
- Se termine par `InetAddress.getByName()`⁵
- Le reste se fait principalement dans la JNI
- On ajoute le dernier comme puit d'exploitation

// pseudocode

```
if (className.equals("java/net/InetAddress")
    && methodName.equals("getByName")
    && argIndex == 0)
    return true;
```

⁵(Code - `java.net.InetAddress`)[<https://github.com/openjdk/jdk/blob/412e306d81209c05f55aee7663f7abb80286e361/src/java.base/share/classes/etAddress.java>]

La démonstration

InetAddress.getByName()

- URLLDNS est bien identifiée
- Une chaine inconnue, avec JMXServiceURL, l'est aussi

```
java/net/URL.hashCode()I (0)
java/net/URLStreamHandler.hashCode(Ljava/net/URL;)I (1)
java/net/URLStreamHandler.getHostAddress(Ljava/net/URL;)Ljava/net/InetAddress; (1)
java/net/URL.getHostAddress()Ljava/net/InetAddress; (0)
java/net/InetAddress.getByName(Ljava/lang/String;)Ljava/net/InetAddress; (0)

javax/management/remote/JMXServiceURL.readObject(Ljava/io/ObjectInputStream;)V (1)
javax/management/remote/JMXServiceURL.validate(Ljava/lang/String;Ljava/lang/String;ILjava/lang/String;)V (2)
javax/management/remote/JMXServiceURL.validateHost(Ljava/lang/String;I)V (0)
java/net/InetAddress.getByName(Ljava/lang/String;)Ljava/net/InetAddress; (0)
```

- Le nom d'hôte doit être une adresse IPv6 numérique⁶
 - Doit être entouré de [et]
 - Doit contenir :

```
if (isNumericIPv6Address(h)) {  
    /* We assume J2SE >= 1.4 here.  Otherwise you can't  
       use the address anyway.  We can't call  
       InetAddress.getByName without checking for a  
       numeric IPv6 address, because we mustn't try to do  
       a DNS lookup in case the address is not actually  
       numeric.  */  
    try {  
        InetAddress.getByName(h);
```

- Inatteignable⁷ en pratique

⁶Implémentation - JMXServiceURL

⁷Jusqu'à preuve du contraire

⁸Spécification - JMXServiceURL

La conclusion

- Ajout de sources de teinte
 - `ObjectInputStream.readUTF()`
 - `ObjectInputStream.read()`
 - `ObjectInput.readObject()`
- Ajout de sources de désérialisation
 - `Externalizable.readExternal`
- Ajout de puits d'exploitation
 - `InetAddress.getByName()`
- Ajout d'un support pour les primitifs
 - Limité, grâce à la source de teinte `read()`

D'autres éléments

- Difficulté à tracer les attributs d'instance
- Difficulté à tracer les types primitifs
- Peu d'intérêts à tracer d'autres primitifs s'ils n'ont pas d'influence
- D'autres puits d'exploitation absents:
 - e.g. `InetAddress.getAllByName()`
- Les autres points déjà mentionnées:
 - Les appels via la JNI
 - Limité à Java version 8 et inférieure

D'autres éléments

- `hashCode()` comme puit donne des résultats intéressants
 - des capsules possibles pour d'autres gadgets
 - pas assez de temps de creuser durant le projet
- Code plutôt difficile à travailler
- Approche problématique
 - Analyse axée autour des variables de pile et aux arguments
 - Le graphe de flot devrait lier la pile, les attributs et les arguments

présentement

`arg -> pile -> arg`

aussi, idéalement

`arg -> pile -> attr -> pile -> arg`

Questions?

Merci pour votre écoute!