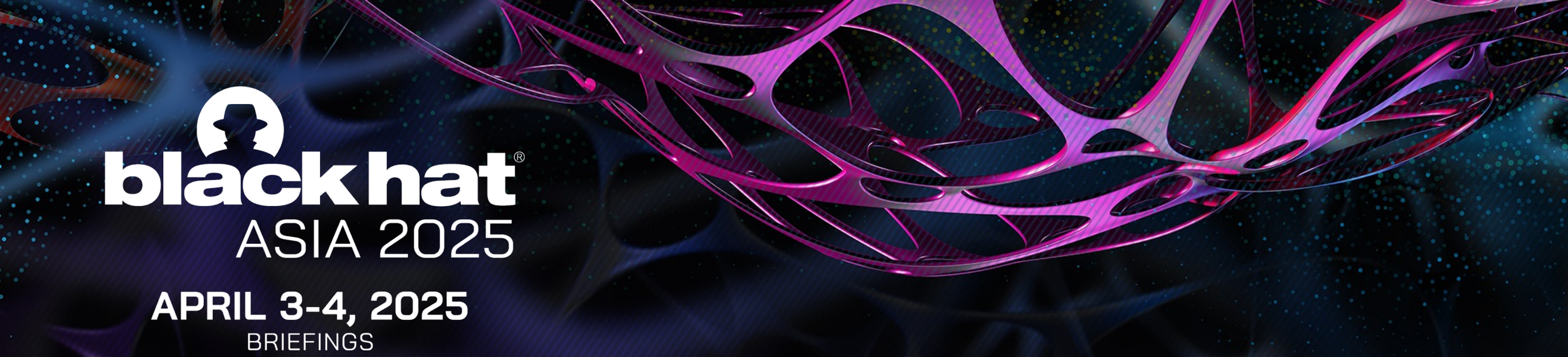


JDD: In-depth Mining of Java Deserialization Gadget Chain via Bottom-up Gadget Search and Dataflow aided Payload Construction

Speaker: Bofei Chen, Yinzhi Cao
Other Contributors: Lei Zhang, Xinyou Huang, Yuan Zhang, Min Yang

Who Are We

Bofei Chen (Speaker)

- PhD student at Fudan University @ Secsys Lab
- Focus on program analysis, vulnerability detection and exploitation.

Yinzhi Cao (Speaker)

- Associate Professor at Johns Hopkins University
- Technical Director at the JHU Information Security Institute
- Focus on security and privacy of the Web, smartphones, and machine learning using program analysis techniques.

Who Are We

Lei Zhang

- Assistant Professor at Fudan University @ Secsys Lab
- Focus on vulnerability detection, exploitation, and automatic fixes, etc.

Xinyou Huang

- Master student at Fudan University @ Secsys Lab
- Focus on dynamic and static program analysis, vulnerability exploitation.

Yuan Zhang

- Professor at Fudan University @ Secsys Lab (co-director)
- Focus on vulnerability research (e.g., Web, agents, kernel and firmware)

Min Yang

- Professor at Fudan University @ Secsys Lab (leader)
- Focus on vulnerability discovery, mitigation, and privacy protection, etc.

Agenda

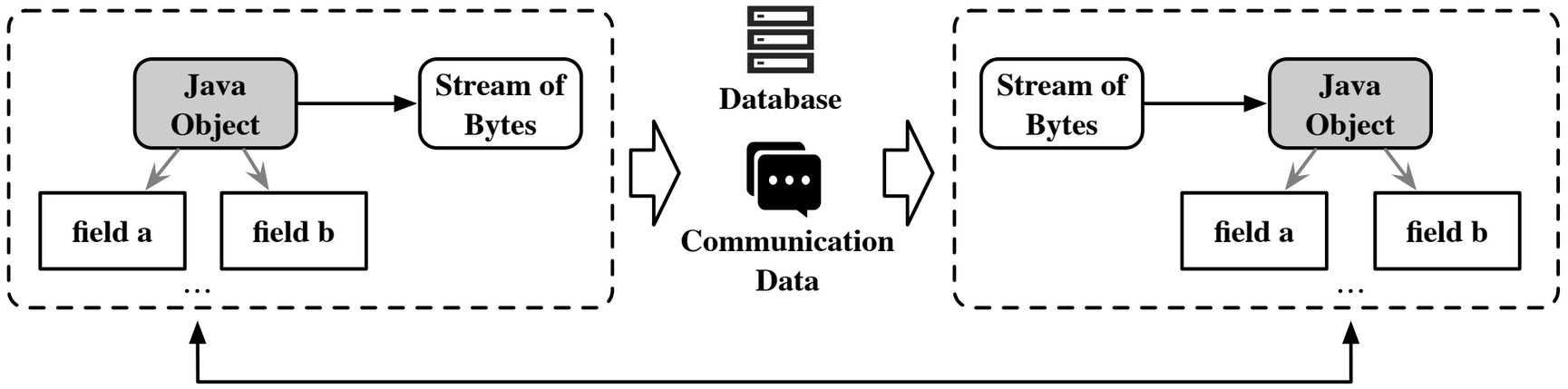
- Introduction
- Technique Challenges
- JDD: Approach and Implementation
- Evaluation and New Findings
- Conclusion & Takeaways

Introduction

- What is a Java deserialization vulnerability?
- Why is Java deserialization vulnerability worth researching?
- How to detect and exploit a Java deserialization vulnerability?
- Mitigation and discussion.

Java Serialization and Deserialization

- Serialization and deserialization are inverse processes of each other. An object's fields are preserved along with their assigned values.



Application Scenario

- Communication
- Persistence
- Data Exchange Format
- Caching
- …

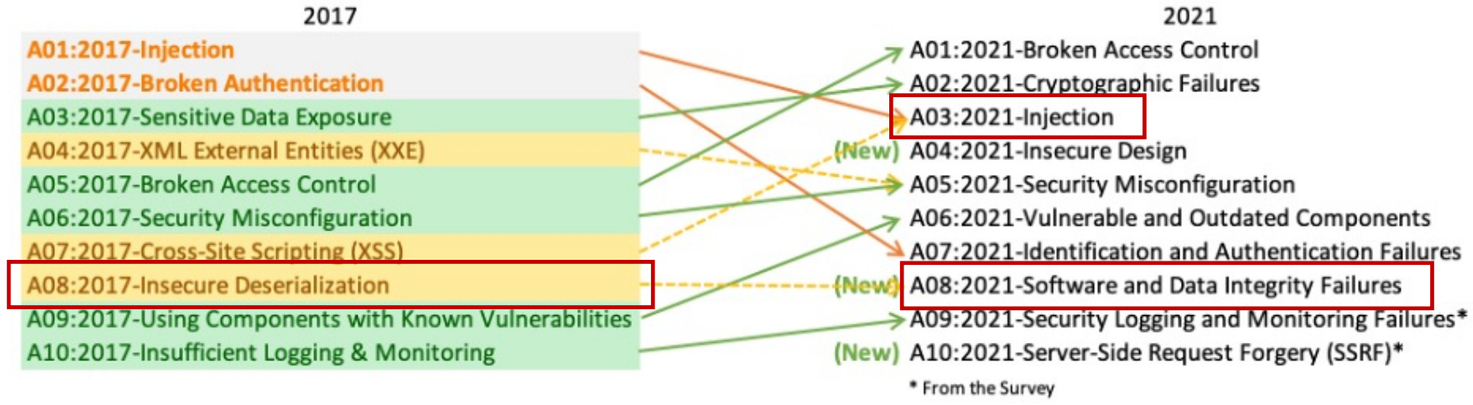
Java Deserialization Vulnerability

- Serialization and deserialization are inverse processes of each other. An object's fields are preserved along with their assigned values.

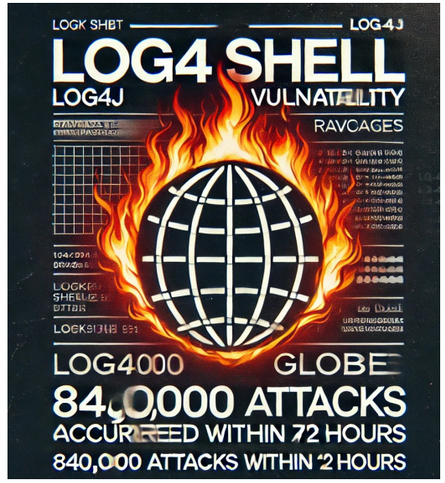
➔ **By carefully manipulating the types and values of serialized data, an attacker can control the deserialization process**, potentially leading to remote code execution or other severe security impacts.

Why is Java Deserialization vulnerability worth researching?

- **High-impact security risks**
  - Can achieve attack consequences such as ***Remote Code Execution (RCE),*** data tampering, Denial of Service (DoS)…



Java deserialization vulnerabilities rank among the Top 10 in OWASP



Log4Shell "nuclear bomb vulnerability" (CVE-2021-44228)

Why is Java Deserialization vulnerability worth researching?

- **Widespread use of deserialization**

- The built-in serialization/deserialization mechanism in Java is widely integrated across multiple frameworks, libraries and features (e.g., RMI, HTTP sessions…).

- Thus, completely avoiding or replacing it can be highly challenging.

Java Deserialization Vulnerability

```
1  // Client Side
2  // For example, a request message
3   Object message = getRequestMessage();
4  // Serialize the Java object by Hessian protocol
5   byte[] serializedData = hessianSerialize(message)
6  // Send the serialized data to the server
7   Socket socket = new Socket(host_of_victim_server, port)
8   Socket.getOutputStream().write(serializedData).flush()
```

Send the serialized data to the target server.

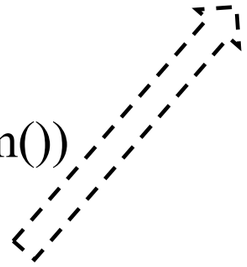
Java Deserialization Vulnerability

```
1  // Client Side
2  // For example, a request message
3   Object message = getRequestMessage();
4  // Serialize the Java object by Hessian protocol
5   byte[] serializedData = hessianSerialize(message)
6  // Send the serialized data to the server
7   Socket socket = new Socket(host_of_victim_server, port)
8   Socket.getOutputStream().write(serializedData).flush()
```

```
1  // Server Side
2  // Receive the serialized data from the client
3   ServerSocket serverSocket = new ServerSocket(port);
4   Socket socket = serverSocket.accept();
5   Hessian2Input hi = new Hessian2Input(socket.getInputStream())
6  // Deserialize the received serialized data
7   Message deserMsg = (Message) hi.readObject();
```

- Reconstruct the original *Message* object

- Use the reconstructed object in the system's business logic (e.g., message handling, order processing).



Receive the serialized data and deserialize it into a Java object

Java Deserialization Vulnerability

```
1  // Client Side
2  // For example, a well-crafted HashMap instance
3   Object hashMap = getRequestMessage();
4  // Serialize the Java object by Hessian protocol
5   byte[] serializedData = hessianSerialize(hashMap)
6  // Send the serialized data to the server
7   Socket socket = new Socket(host_of_victim_server, port)
8   Socket.getOutputStream().write(serializedData).flush()
```

```
1  // Server Side
2  // Receive the serialized data from the client
3    ServerSocket serverSocket = new ServerSocket(port);
4    Socket socket = serverSocket.accept();
5    Hessian2Input hi = new Hessian2Input(socket.getIn   tStream())
6  // Deserialize the received serialized data
7   Message deserMsg = (Message) hi.readObject();
```

E.g., a well-crafted HashMap instance

```
public class MapDeserializer {
    Object readMap(A…HessianInput in)
{… Map map = new HashMap();
      while(!in.isEnd()){
       map.put(in.readObject(), …); // entry
    }}
}
```

Receive the serialized data and deserialize it into a Java object

Java Deserialization Vulnerability

```
1  // Client Side
2  // For example, a well-crafted HashMap instance
3   Object hashMap = getRequestMessage();
4  // Serialize the Java object by Hessian protocol
5   byte[] serializedData = hessianSerialize(hashMap)
6  // Send the serialized data to the server
7   Socket socket = new Socket(host_of_victim_server, port)
8   Socket.getOutputStream().write(serializedData).flush()
```

```
1  // Server Side
2  // Receive the serialized data from the client
3    ServerSocket serverSocket = new ServerSocket(port);
4    Socket socket = serverSocket.accept();
5    Hessian2Input hi = new Hessian2Input(socket.getIn   tStream())
6  // Deserialize the received serialized data
7   Message deserMsg = (Message) hi.readObject();
```

E.g., a well-crafted HashMap instance

```
public class MapDeserializer {
   Object readMap(A…HessianInput in)
{… Map map = new HashMap();
     while(!in.isEnd()){
      map.put(in.readObject(), …); // entry
   }}
}

public class HashMap {
   Node<K,V>[] table;          // Entry method
   public void put(K key, V value){…
  1 key.equals(value); … }}
```

Receive the serialized data and deserialize it into a Java object

Java Deserialization Vulnerability

```
public class HashMap {
    Node<K,V>[] table;
    public void put(K key, V value){…
  1 key.equals(value); … }}
public class EvilExample{
    public String cmd;
    public boolean equals(Object o){…
      Runtime.getRuntime()
                  .exec((EvilExample)o.cmd);}}
                                    2
```

① Control the **type** of *key*: control the deserialization process to execute the **EvilExample.equals** method.

Java Deserialization Vulnerability

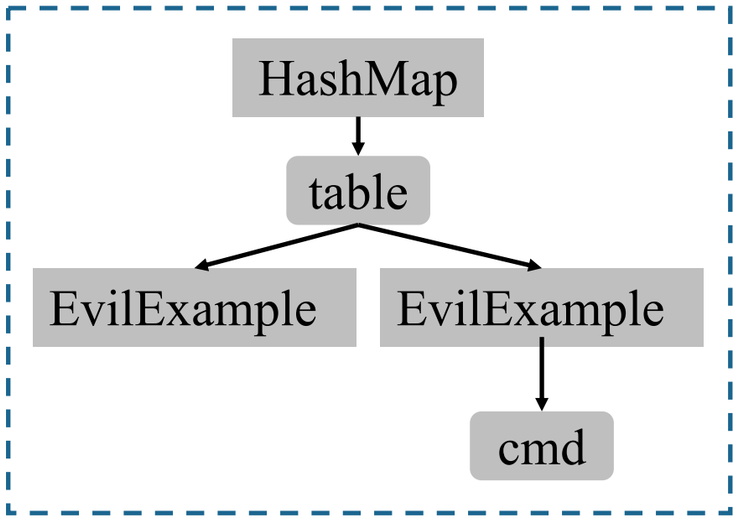
```
public class HashMap {
    Node<K,V>[] table;
    public void put(K key, V value){…
  1 key.equals(value); … }}
public class EvilExample{
    public String cmd;
    public boolean equals(Object o){…
      Runtime.getRuntime()
                  .exec((EvilExample)o.cmd);}}
                                    2
```

Remote Code Execution

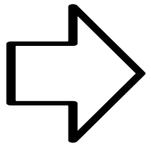
① Control the **type** of *key*: control the deserialization process to execute the **EvilExample.equals** method.
② Control the **value** of *o.cmd*: control the executed code.

How to detect and exploit a Java Deserialization vulnerability?

- **Gadget Chain**: A chain of **internal Java methods** (i.e., **gadgets**) that can invoke security-sensitive method(s) capable of executing malicious code during the deserialization process.
- **Injection Object**: A **serialized object** that drives the execution of the gadget chain.



Serialized => Injection Object

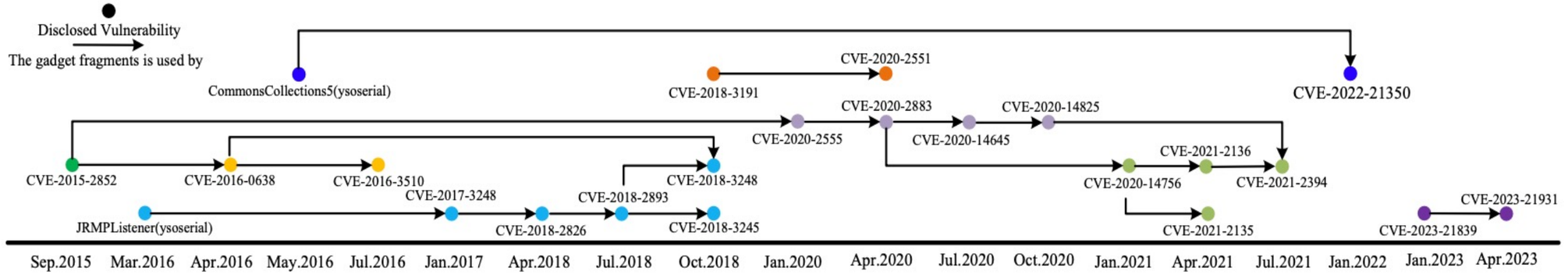


E.g., HashMap.put
-> EvilExample.equals
-> Runtime.exec

Gadget Chain

Mitigation and Discussion

- **Commonly used defenses**
  - Setting a black/whitelist to restrict classes that can be deserialized to truncate the gadget chains
  - Restricted blocking options: to ensure that normal business functions are not affected.



WebLogic's JOI vulnerabilities and the **reuse of their gadgets**, which lead to the **incomplete patch problem**

Mitigation and Discussion

- An example of reusing partial gadgets to generate a new exploitable gadget chain.

```
1  // A part of code of the patch of CVE-2020-2883.
2  // Rewriting resolveClass method of ObjectInputStream.
3  Class<?> resolveClass(ObjectStreamClass desc) {
4      String clzName = desc.getName();
5      if (this.blackList.contains(clzName)) {
6          throw new  InvalidClassException();
7      }
8      return super.resolveClass(desc);
9  }
10 String[] blackList = {
11 “com.tangosol.util.extractor.ReflectionExtractor”,
12 “com.tangosol.util.extractor.MultiExtractor” ...
13 };
```

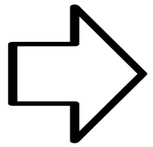
```
1  PriorityQueue#readObject
2    ↳PriorityQueue#heapify
3      ↳PriorityQueue#siftDown
4        ↳PriorityQueue#siftDownUsingComparator
5          ↳AbstractExtractor#compare
6            ↳MultiExtractor#extract
7              ↳ReflectionExtractor#extract
8                ↳Method#invoke
```

⇩ CVE-2020-14645

```
1  ExtractorComparator#compare
2    ↳UniversalExtractor#extract
3      ↳UniversalExtractor#extractComplex
```

Mitigation and Discussion

Persistence of the threat



- Attackers can find replaceable gadgets that **bypass defenses** (e.g., blacklist).
  - **Java's dynamic features**
  - **Widespread use of third-party components**

- The fundamental design of Java deserialization allows for a large attack surface, and new classes with exploitable features may be introduced over time.

Technical Challenges

- How to detect gadget chains?
- How to generate the injection object?

Question I: How to detect Gadget Chains in the real-world?

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash & p.key != key & key != null)
8     as1 key.equals(p.key); }…}
9  }                          as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12    private final K key; private V value;                    Candidates: 2751
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                 xstrFSB          jsonObj
```

…

```
39 /* Gadget Fragment VI: ObjectWriter2.write -> FieldWriter.write */
40 public class ObjectWriter2<T> {                          Candidates: 87
41    public final FieldWriter fieldWriter;
42    void write(…, Object object, ...) { fieldWriter.write(…, object);}
43 }
44 /* Gadget Fragment VII: FieldWritterObject.write -> Method.invoke */
45 abstract class FieldWriterObject<T> {
46    // the method to get the value of a field. E.g. getter method  Candidates: 55
47    public final Method method;
48    public boolean write(…, T object) { ...getFieldValue(object);}
49    public Object getFieldValue(Object object) {this.method.invoke(object); ...}
50 }                                  // Unsafe Reflection
                                                    jsonObj.map
```

Part of the simplified exploitable Gadget Chain detected by JDD

An deserialization entry method (i.e., source)

Question I: How to detect Gadget Chains in the real-world?

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash & p.key != key & key != null)
8     as1 key.equals(p.key); }…}
9  }                          as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12    private final K key; private V value;                    Candidates: 2751
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                 xstrFSB          jsonObj
```

•••

```
39 /* Gadget Fragment VI: ObjectWriter2.write -> FieldWriter.write */
40 public class ObjectWriter2<T> {                          Candidates: 87
41    public final FieldWriter fieldWriter;
42    void write(…, Object object, ...) { fieldWriter.write(…, object);}
43 }
44 /* Gadget Fragment VII: FieldWritterObject.write -> Method.invoke */
45 abstract class FieldWriterObject<T> {
46    // the method to get the value of a field. E.g. getter method  Candidates: 55
47    public final Method method;
48    public boolean write(…, T object) { ...getFieldValue(object);}
49    public Object getFieldValue(Object object) {this.method.invoke(object); ...}
50 }                                  // Unsafe Reflection
                                                    jsonObj.map
```

Part of the simplified exploitable Gadget Chain detected by JDD

An deserialization entry method (i.e., source)

A controllable dynamic method call

Question I: How to detect Gadget Chains in the real-world?

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash & p.key != key & key != null)
8     as1 key.equals(p.key); }…}
9  }                          as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12    private final K key; private V value;                    Candidates: 2751
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                 xstrFSB          jsonObj
```

…

```
39 /* Gadget Fragment VI: ObjectWriter2.write -> FieldWriter.write */
40 public class ObjectWriter2<T> {                          Candidates: 87
41    public final FieldWriter fieldWriter;
42    void write(…, Object object, ...) { fieldWriter.write(…, object);}
43 }
44 /* Gadget Fragment VII: FieldWritterObject.write -> Method.invoke */
45 abstract class FieldWriterObject<T> {
46    // the method to get the value of a field. E.g. getter method  Candidates: 55
47    public final Method method;
48    public boolean write(…, T object) { ...getFieldValue(object);}
49    public Object getFieldValue(Object object) {this.method.invoke(object); ...}
50 }                                  // Unsafe Reflection
                                                    jsonObj.map
```

An deserialization entry method (i.e., source)

A controllable dynamic method call

unsafe Reflection

Part of the simplified exploitable Gadget Chain detected by JDD

Question I: How to detect Gadget Chains in the real-world?

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash & p.key != key & key != null)
8     as1 key.equals(p.key); }…}
9  }                          as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{                    Candidates: 2751
12    private final K key; private V value;
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                  xstrFSB          jsonObj
```

…

```
39 /* Gadget Fragment VI: ObjectWriter2.write -> FieldWriter.write */
40 public class ObjectWriter2<T> {                          Candidates: 87
41    public final FieldWriter fieldWriter;
42    void write(…, Object object, ...) { fieldWriter.write(…, object);}
43 }
44 /* Gadget Fragment VII: FieldWritterObject.write -> Method.invoke */
45 abstract class FieldWriterObject<T> {
46    // the method to get the value of a field. E.g. getter method  Candidates: 55
47    public final Method method;
48    public boolean write(…, T object) { ...getFieldValue(object);}
49    public Object getFieldValue(Object object) {this.method.invoke(object); ...}
50 }                                  // Unsafe Reflection
                                                    jsonObj.map
```

Part of the simplified exploitable Gadget Chain detected by JDD

An deserialization entry method (i.e., source)

A controllable dynamic method call

```
52 /* Gadget Fragment VIII: Serve…Impl.get…vers ->Runtime.exec */
53 public class ServerManagerImpl …{
54    HashMap serverTable;
55    public int[] getActiveServers(){…
56    (ServerTableEntry)serverTable.get(key).isValid()…}
57 public class ServerTableEntry{
58    private String activationCmd;
59    synchronized boolean isValid(){
60       if ((state == ACTIVATING) || (state == HELD_DOWN)) return true;
61       if (state == ACTIVATED) {
62       if (activateRetryCount < ActivationRetryMax) {
63          activate();…}}
64    synchronized void activate(){…Runtime.getRuntime().exec(activationCmd);}
65 }                                                  // Command Injection Attack
```

unsafe Reflection

Command Injection Attack (i.e., sink)

Challenge I: Static Path Explosion

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash && p.key != key & key != null)
8           as1 key.equals(p.key); } ...}                                  ①
9                                  as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{                    Candidates: 2751
12    private final K key; private V value;
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                    xstrFSB              jsonObj
18 public class AbstractMap implements …{
19    private static boolean eq(Object o1, Object o2) {
20       return o1 == null ? o2 == null : o1.equals(o2);}}                  ②
21 /* Gadget Fragment III: XStringForFSB.equals -> Object.toString */
22 public class XStringForFSB extends …{
23    protected Object m_obj;                          Candidates: 2751
24    public boolean equals(Object obj2){
25       if (null != obj2 && !(obj2 instanceof XNumber)…)
26          return equals(obj2.toString());…}}                              ③
27                        jsonObj
28 /* Gadget Fragment IV: JSONObject.toString -> JSONWriter.write */
29 public abstract class JSONObject {
30    Node<String, Object> table;                      Candidates: 3129
31    public String toString() {
32       (JSONWriter)writer.write(this); ...}}                              ④
33 /* Gadget Fragment V: JSONWriterUTF16.write ->ObjectWriter.write */
34 class JSONWriterUTF16 extends JSONWriter  {          Candidates: 3
35    public final void write(JSONObject jsonObject){ ...
36       for (Object value: jsonObject.map.values()){
37          objectWriter.write(this, value, (Object)null, (Type)null, 0L);}}  ⑤
38                              jsonObj.map
39 /* Gadget Fragment VI: ObjectWriter2.write -> FieldWritter.write */
40 public class ObjectWriter2<T> {                      Candidates: 87
41    public final FieldWriter fieldWriter;
42    void write(…, Object object, ...) { fieldWriter.write(…, object);}    ⑥
43 }
44 /* Gadget Fragment VII: FieldWritterObject.write -> Method.invoke */
45 abstract class FieldWriterObject<T> {
46    // the method to get the value of a field. E.g. getter method  Candidates: 55
47    public final Method method;
48    public boolean write(…, T object) { ...getFieldValue(object);}
49    public Object getFieldValue(Object object) {this.method.invoke(object); ...}
50 }                                  // Unsafe Reflection
                                                          jsonObj.map
```

 # Fragment 1
HashMap.put (head)
HashMap.putVal
Object.equals (end)

Candidate: 2751

 # Fragment 2
Abstract…Entry.equals (head)
AbstractMap.access$000
AbstractMap.eq
Object.equals (end)

Candidate: 2751

 # Fragment 3
XStringForFSB.equals
…

Candidate: 3129

 # Fragment 4
JSONObject.toString
…

Candidate: 3

 # Fragment 5
JSONWriterUTF16.write

Candidate: 87

 # Fragment 6
ObjectWriter2.write

Candidate: 55

 # Fragment 7
Field…ethod.write (head)
Field…ethod.getFieldValue (head)
Method.invoke (end)

 # Fragment 8
ServerM….getActiveServers (head)
ServerTableEntry.isValid
ServerTableEntry.activate
Runtime.exec (sink)

- During the search, it is easy to detect many dynamic method calls that the attacker can control.
- **Top-down** candidate search methods could grow exponentially with the search length.

Challenge II: Complex Object Field Relations

 # Fragment 1
HashMap.put
HashMap.putVal

 # Fragment 2
AbstractMap$SimpleEntry.equals
AbstractMap.access$000
AbstractMap.eq

 # Fragment 3
XStringForFSB.equals

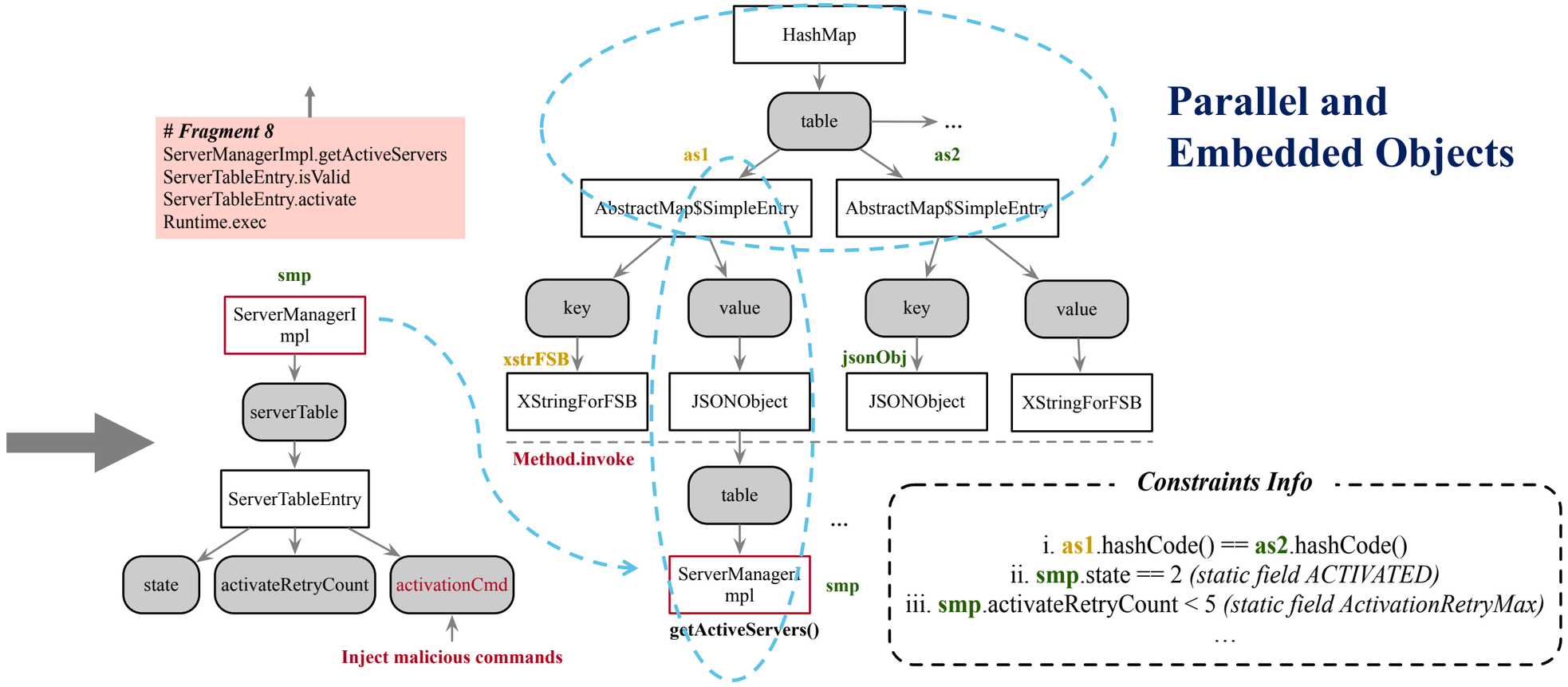
 # Fragment 4
JSONObject.toString

 # Fragment 5
JSONWriterUTF16.write

 # Fragment 6
ObjectWriter2.write

 # Fragment 7
FieldWriterObjectMethod.getFieldValue
Method.invoke

 # Fragment 8
ServerManagerImpl.getActiveServers
ServerTableEntry.isValid
ServerTableEntry.activate
Runtime.exec



Parallel and Embedded Objects

Challenge II: Complex object field relations

- Parallel and Embedded Injection Object Structure.
- Dependencies and constraints between fields.

JDD: Approach and Implementation

- Fragment-based Summary and Bottom-up Gadget Chain Search
- Dataflow-aided Injection Object Construction

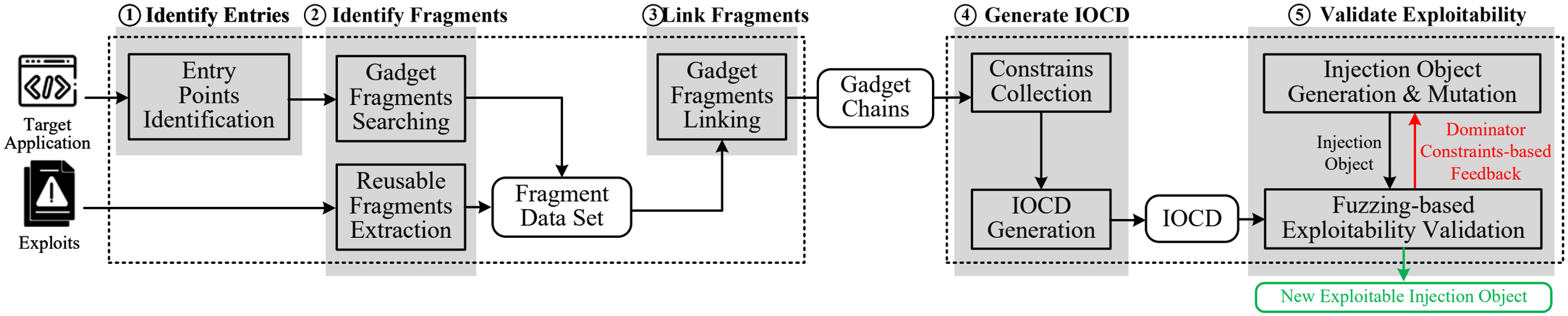
Key Ideas

- **Path Explosion challenge**: **fragment-based summary** and **bottom-up** search approach.
  - *Key Observation: a bottom-up search reduces maximum static search time from exponential to polynomial, i.e., from $O(eM^n)$ to $O(n^3 M^2 + enM)$.*

- **Complex Object Field Relations:** use static taint analysis to **construct dataflow dependencies between possible injection objects' fields** and use them to guide dynamic fuzzing to generate exploitable objects.
  - *Key Observation: different injection objects, e.g., their fields, are connected via dataflows.*

Overall Architecture

- **Stage I: Gadget Chain Detection**
  - Identify Entry Points
  - Search Fragments
  - Link Fragments via a bottom-up approach
- **Stage II: Injection Object Generation**
  - Generate IOCD
  - IOCD-enhanced directional Fuzzing to verify the exploitability of gadget chains



Fragment-based Summary

Q: What is the biggest "culprit" that leads to path explosion in static analysis?

A: Dynamic method invocation

Fragment-based Summary

Q: What is the biggest "culprit" that leads to path explosion in static analysis?

A: Dynamic method invocation

- **Break down the one-time search for a complete gadget chain into the search and chaining of multiple smaller and simpler segments based on dynamic method calls.**
- **Generate bottom-up summaries for each segment to minimize redundant analysis**

Fragment-based Summary

Q: What is the biggest "culprit" that leads to path explosion in static analysis?

A: Dynamic method invocation

- **Break down the one-time search for a complete gadget chain into the search and chaining of multiple smaller and simpler segments based on dynamic method calls.**
- **Generate bottom-up summaries for each segment to minimize redundant analysis**

Q: Why not generate detailed summaries for each method directly?

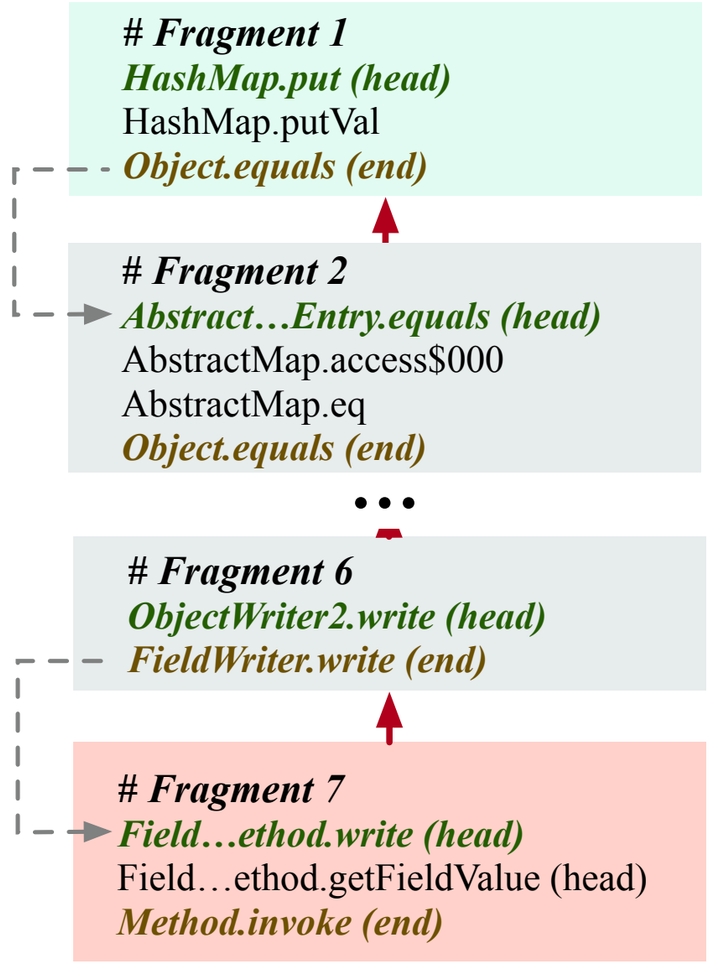
A: To balance path explosion and state explosion.

Component of Gadget Fragment

Head: entry method
- Source
- Exist some dynamic methods invocations that could jump to it

End: exist method
- Dynamic method invocation or security-sensitive method.

Other gadgets: non-dynamic methods to connect the *head* and *end*.



Types of Gadget Fragment

Source Fragment

- whose head is a source method (e.g., readObject/ Map.put).

```
# Fragment 1
HashMap.put (head)
HashMap.putVal
Object.equals (end)
```

Free-State Fragment

- chains the execution sequence between two dynamic method invocations.

```
# Fragment 2
Abstract…Entry.equals (head)
AbstractMap.access$000
AbstractMap.eq
Object.equals (end)
```

Sink Fragment

- whose end is a sink.

```
# Fragment 7
Field…ethod.write (head)
Field…ethod.getFieldValue (head)
Method.invoke (end)
```

Summarized Information

Bottom-up taint behavior: dataflow reachability of the gadget chain

- Parameter taint relationships from *End* (e.g., equals) to *Head* (e.g., put)

Linking Condition: control flow reachability of the gadget chain

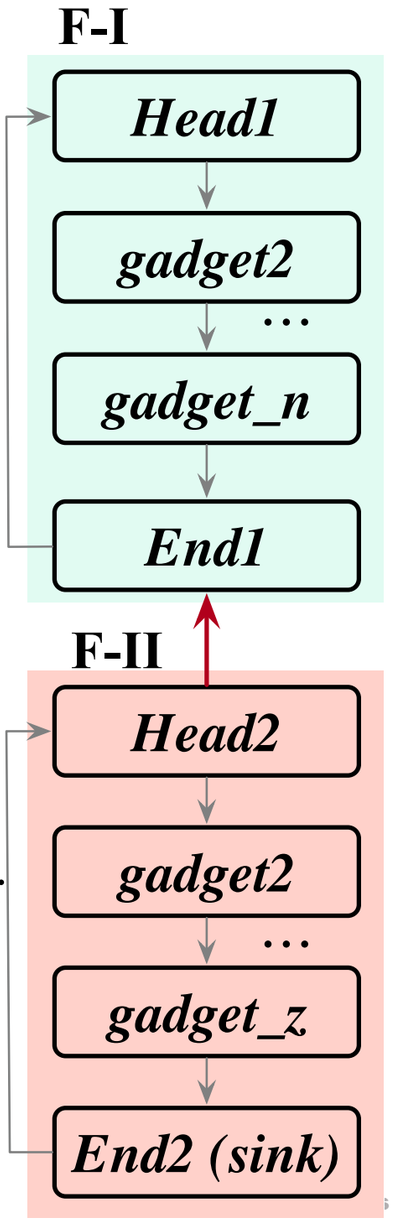
- The methods that the end gadget in this fragment can jump to. (Vary slightly for different types of dynamic invocations)
- E.g., *Head* of F-II need to be the overridden method of *End* of F-I

Exploit Condition: the specific exploit condition for the sink gadget (in Sink Fragment)

(a) Taint behavior:
End pi => Head p[x,y,…]

(b) Link condition:
E.g., Head2 is a overridden of Head1

(c) Exploit Condition:
E.g., in-coming parameters[1,2] of End2 need to be tainted



STEP 1: Identify Entry Points

Step 1: Identify the entry points of deserialization (i.e., sources)

- Extract and filter deserialization entry methods (***i.e., sources***)

Protocol	Entry Points	Supported Dynamic Feature	Unserializable Class Support
JDK	readObject() readObjectNoData() readResolve() readExternal()	Polymorphism Reflection Proxy	NO
T3/IIOP	readObject() readObjectNoData() readResolve() readExternal()	Polymorphism Reflection Proxy	NO
Hessian	Map.put() toString()	Polymorphism Reflection	YES
Hessian-lite [7]	Map.put()	Polymorphism Reflection	NO
Hessian-sofa [8]	Map.put() toString()	Polymorphism Reflection	YES
XStream	readObject() Map.put()	Polymorphism Reflection Proxy	YES

A deserialization entry method (i.e., source)

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3     Node<K,V>[] table;
4     V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5     V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6        Node<K,V> p = table[pre_index]; // p is an element in table
7        if (p.hashCode() == hash & p.key != key & key != null)
8        as1 key.equals(p.key); }…}
9  }                                  as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12    private final K key; private V value;                    Candidates: 2751
13    public int hashCode() {… return key.hashCode()^value.hashCode();}
14    public boolean equals(Object o) {
15       if (o instanceof Map.Entry)
16          return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                  xstrFSB              jsonObj
```

STEP 2: Search Fragments

Step 2: Identify Gadget Fragments with Static Taint Analysis

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3    Node<K,V>[] table;
4    V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5    V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6      Node<K,V> p = table[pre_index]; // p is an element in table
7      if (p.hashCode() == hash & p.key != key & key != null)
8        as1 key.equals(p.key); }…}
9  }                                  as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{                    Candidates: 2751
12   private final K key; private V value;
13   public int hashCode() {… return key.hashCode()^value.hashCode();}
14   public boolean equals(Object o) {
15     if (o instanceof Map.Entry)
16       return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }              xstrFSB          jsonObj
18 public class AbstractMap implements …{
19   private static boolean eq(Object o1, Object o2) {
20     return o1 == null ? o2 == null : o1.equals(o2);}}
21 /* Gadget Fragment III: XStringForFSB.equals -> Object.toString */
22 public class XStringForFSB extends …{
23   protected Object m_obj;                               Candidates: 2751
24   public boolean equals(Object obj2){
25     if (null != obj2 && !(obj2 instanceof XNumber)…)
26       return equals(obj2.toString());…}}
27                   jsonObj
```

① Fragment Summary:
Taint analysis within a fragment…

 # Fragment 1
HashMap.put (head)
HashMap.putVal
Object.equals (end)

② Search for subsequent gadget fragments

 # Fragment 2
Abstract…Entry.equals (head)
AbstractMap.access$000
AbstractMap.eq
Object.equals (end)

STEP 2: Search Fragments

Step 2: Identify Gadget Fragments with Static Taint Analysis

(1) Search Source: HashMap.put

HashMap.put
HashMap.putVal
Object.equals

(2) Generate Fragment

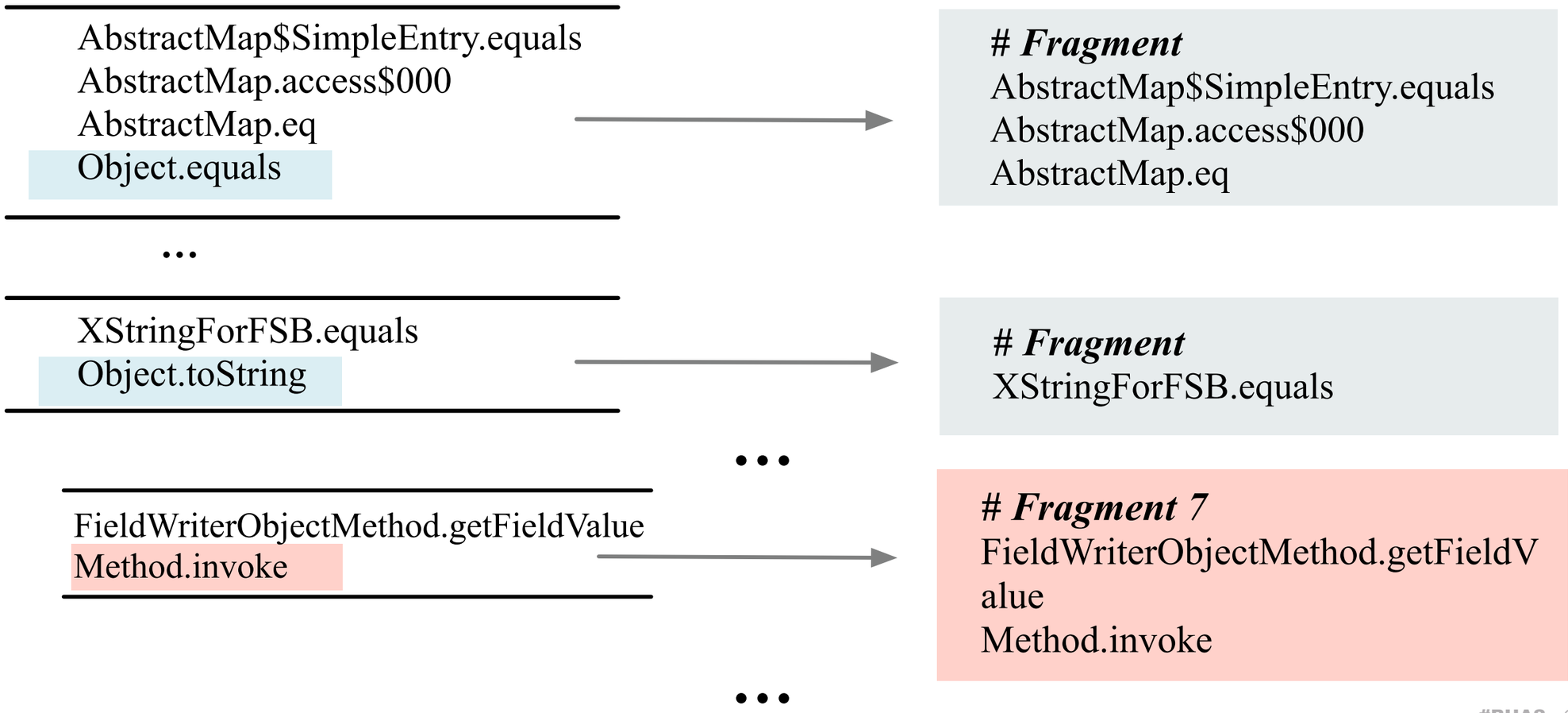
a. Taint Summary
b. Link condition Summary

 # Fragment
head: HashMap.put
HashMap.putVal
end: Object.equals

STEP 2: Search Fragments

Step 2: Identify Gadget Fragments with Static Taint Analysis

(2) Search Sources: methods overwrittern `Object.equals`



STEP 2: Search Fragments

Step 2: Identify Gadget Fragments with Static Taint Analysis

Source Fragments

 # Fragment
head: HashMap.put
HashMap.putVal
end: Object.equals

...

Free-State Fragments

 # Fragment
AbstractMap$SimpleEntry.equals
AbstractMap.access$000
AbstractMap.eq

 # Fragment
XStringForFSB.equals

...

Sink Fragments

 # Fragment 7
FieldWriterObjectMethod.getFieldV
alue
Method.invoke

 # Fragment 8
ServerManagerImpl.getActiveServers
ServerTableEntry.isValid
ServerTableEntry.activate
Runtime.exec

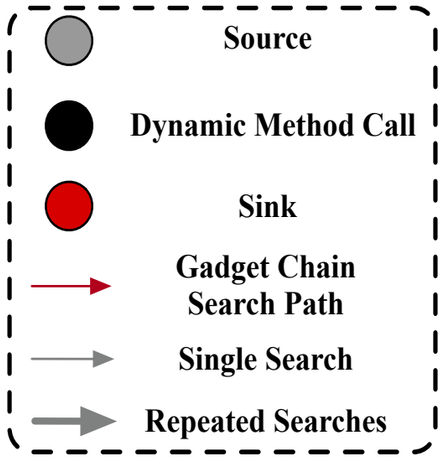
...

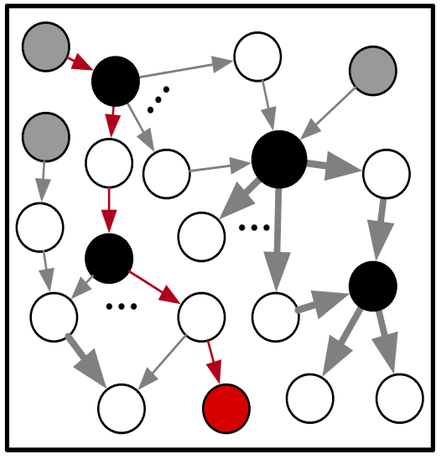
STEP 3: Linking Fragments via a Bottom-up approach

Step 3: Linking Gadget Fragments to Construct Gadget Chains Using a Bottom-up Approach

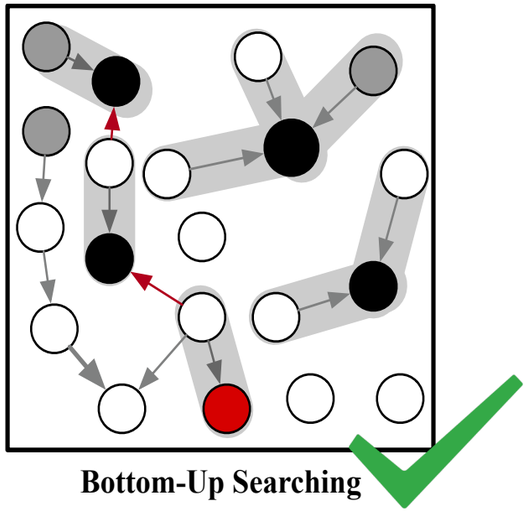
- **Chain gadget fragments from sink to source.**

- Fully reuse existing sink knowledge to minimize repetitive analyses and reduce search complexity.

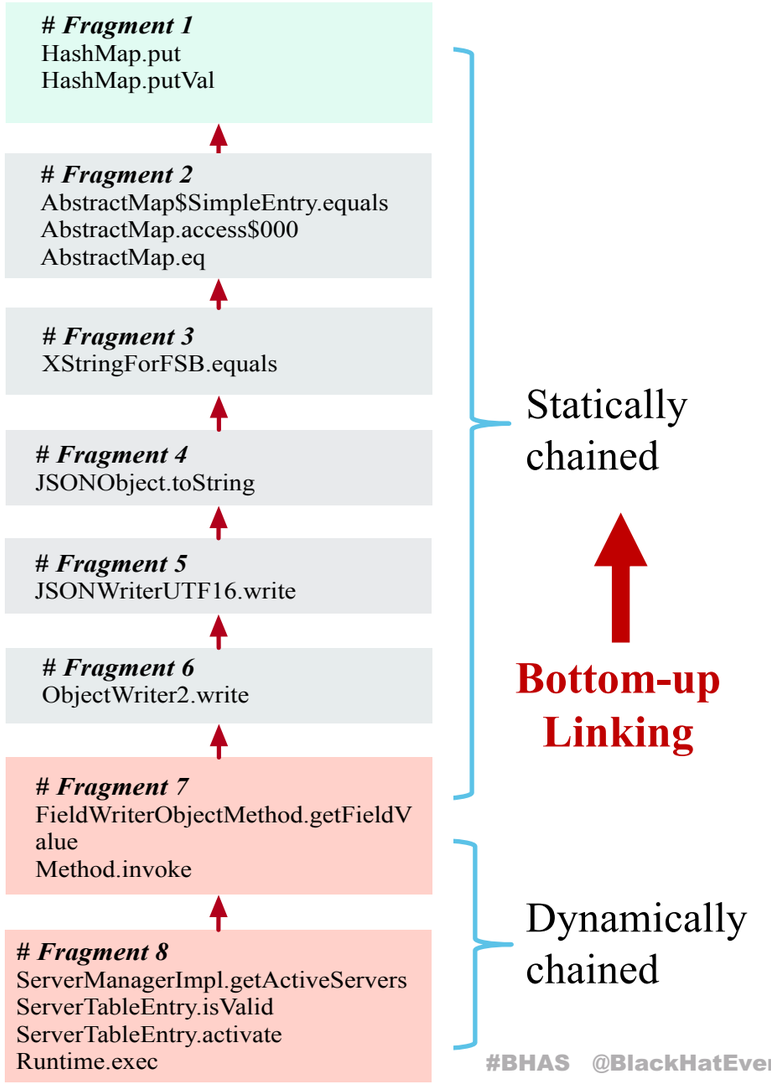




Top-Down Searching

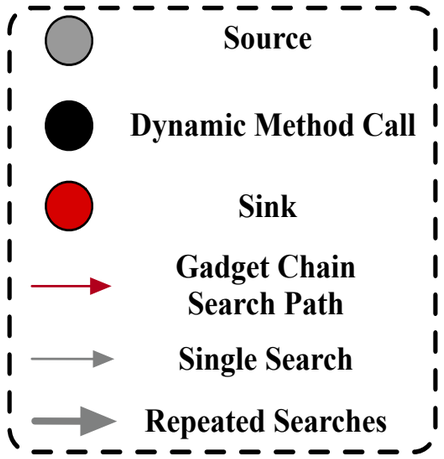


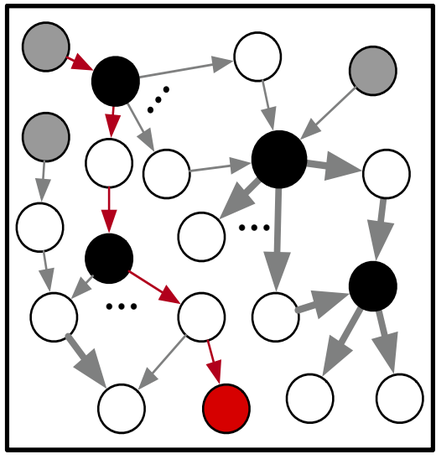
Bottom-Up Searching



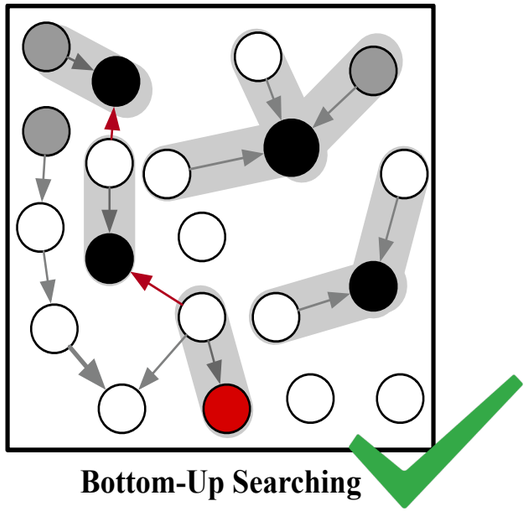
Step 3: Linking Gadget Fragments to Construct Gadget Chains Using a Bottom-up Approach

- **Chain gadget fragments from sink to source.**
- Based on the exploitation conditions of the sink, calculate the precise parameter contamination requirements, etc., for linking.
- Avoid linking calculations for dataflow-unreachable and control-flow-unreachable fragments.

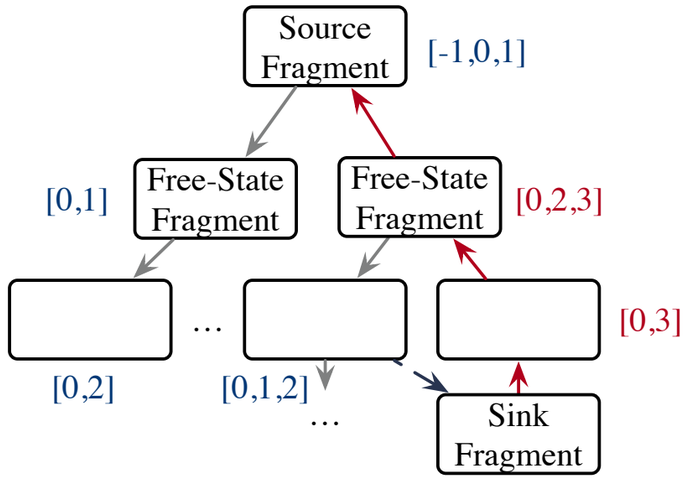




Top-Down Searching



Bottom-Up Searching



Top-Down: Unpredictability to Sink Fragment

required tainted parameters of pre-fragment: [0,3]

STEP 4: IOCD-enhanced Directed Fuzzing

JDD follows the call sequence in the gadget chain to construct dataflow dependencies between possible injection objects' fields as an IOCD
==> To facilitate dynamic fuzzing

- Class hierarchy relationships between object and field instance
- Conditional branches related to fields
- Field dependency constraints
- Fields related to the attack payload

STEP 4: IOCD-enhanced Directed Fuzzing

❖ Class Hierarchy Relationships

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3      Node<K,V>[] table;
4      V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5      V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6          Node<K,V> p = table[pre_index]; // p is an element in table
7          if (p.hashCode() == hash & p.key != key & key != null)
8              as1 key.equals(p.key); }…}
9  }                                  as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12     private final K key; private V value;
13     public int hashCode() {… return key.hashCode()^value.hashCode();}
14     public boolean equals(Object o) {
15         if (o instanceof Map.Entry)
16             return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                      xstrFSB              jsonObj
```

Class hierarchy Relationship

Field Type

Candidates: 2751

- **Taint analysis: for each fragment, which of its fields is link to the next fragment?**
- E.g., “table” field of the HashMap instance (Fragment I) link to Fragment II.

STEP 4: IOCD-enhanced Directed Fuzzing

❖ Class Hierarchy Relationships

```
1  /* Gadget Fragment I: HashMap.put -> HashMap.putVal */
2  public class HashMap implements ...{
3      Node<K,V>[] table;
4      V put(K key, V value) {returen putVal(hash(key), key, value, …);}
5      V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) { ...
6          Node<K,V> p = table[pre_index]; // p is an element in table
7          if (p.hashCode() == hash & p.key != key & key != null)
8          as1 key.equals(p.key); }…}
9  }                                  as2
10 /* Gadget Frafment II: SimpleEntry.equals ->...->Object.equals */
11 public static class SimpleEntry<K,V>{
12     private final K key; private V value;
13     public int hashCode() {… return key.hashCode()^value.hashCode();}
14     public boolean equals(Object o) {
15         if (o instanceof Map.Entry)
16             return eq(key, (Map.Entry)e.getKey()) && eq(value, e.getValue());}
17 }                      xstrFSB              jsonObj
```

Class hierarchy Relationship

Field Type

Candidates: 2751

- **Taint analysis: for each fragment, which of its fields is link to the next fragment?**
  - E.g., “table” field of the HashMap instance (Fragment I) link to Fragment II.
  - Use the head of the subsequent fragment to determine the actual type of the field. E.g., The “table” field stores instances of the SimpleEntry type.

STEP 4: IOCD-enhanced Directed Fuzzing

❖ Conditional Branch & Field Dependence

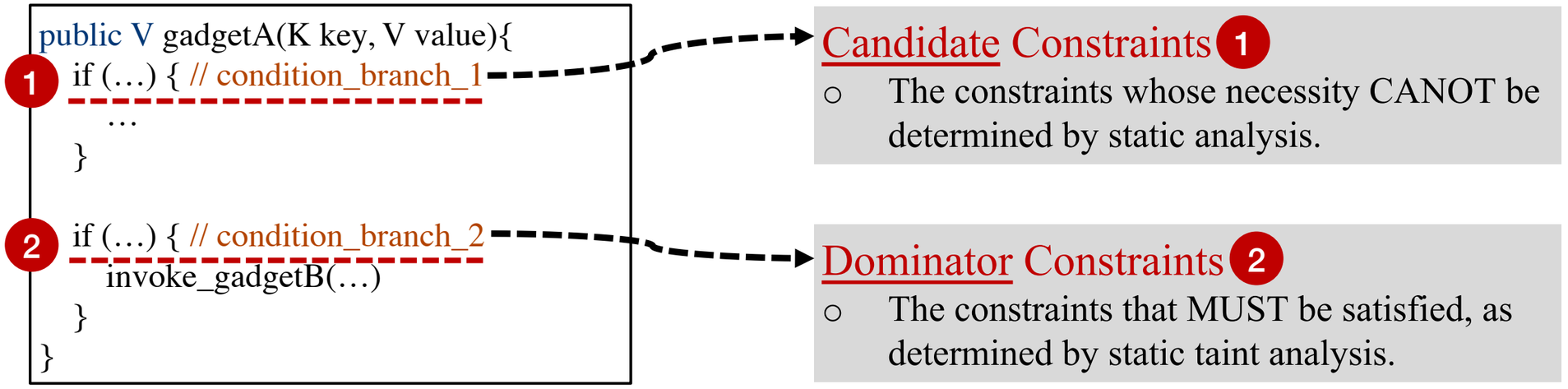
```
57 public class ServerTableEntry{
58     private String activationCmd;
59     synchronized boolean isValid(){
60         if ((state == ACTIVATING) || (state == HELD_DOWN)) return true;
61         if (state == ACTIVATED) {
62         if (activateRetryCount < ActivationRetryMax) {
```

Conditional Branch

- Extract conditional branches related to fields
- Constraint solving

STEP 4: IOCD-enhanced Directed Fuzzing

❖ Dominator Constraints



Two types of constraints that categorized by JDD

- **For <u>Candidate</u> Constraints**

  JDD would ***mutate*** the related fields during the exploration stage of fuzzing.

- **For <u>Dominator</u> Constraints**

  JDD would use the ***constraints solver*** (e.g., Z3) to obtain its concrete value.

STEP 4: IOCD-enhanced Directed Fuzzing

❖ Injection Object Construct Diagram (IOCD)

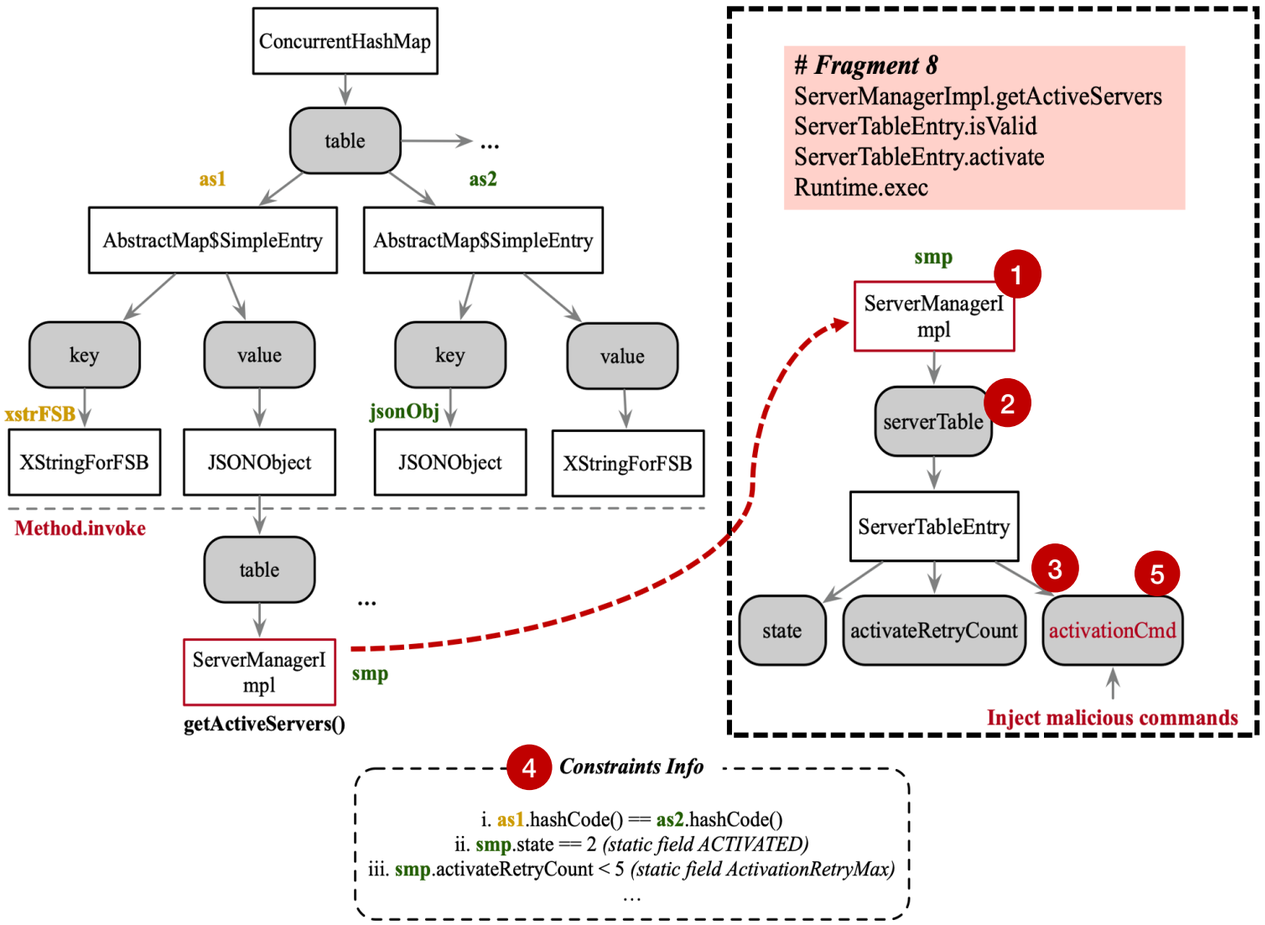


Illustration of IOCD

Definition

- The data structure for describing the Injection Object

Functionality

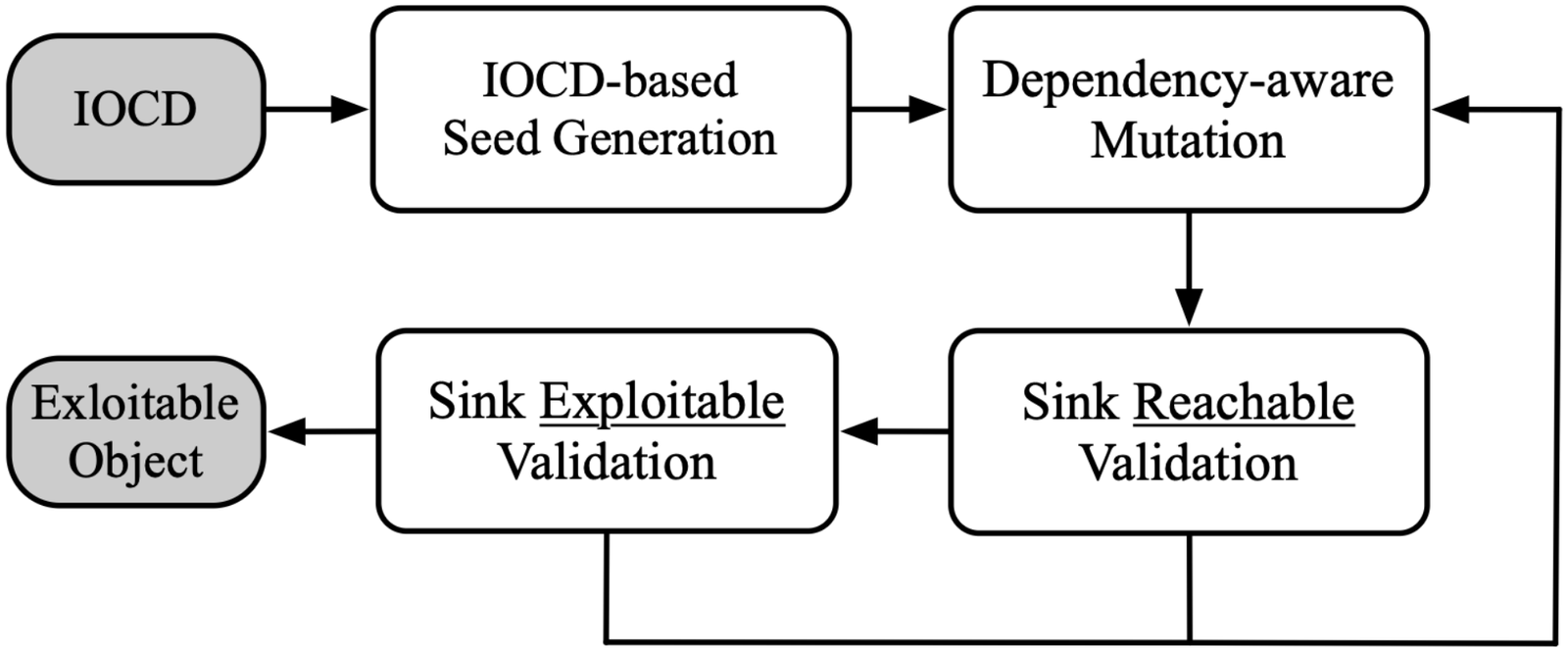
- The Structure of Injection Object
- The Constraints Info of specific fields of Injection Object

Components

- Class-Node 1
- Field-Node 2
- Directed-Edge 3
- Constraints Info (Candidate and Dominator Conditions) 4
- Potential Exploitable Payloads Position 5

STEP5: IOCD-enhanced Directed Fuzzing

❖ Workflow Overview of JDD's Directed Fuzzing



Workflow of JDD's Directed Fuzzing

STEP5: IOCD-enhanced Directed Fuzzing

❖ IOCD-based Seed (Injection Object) Generation

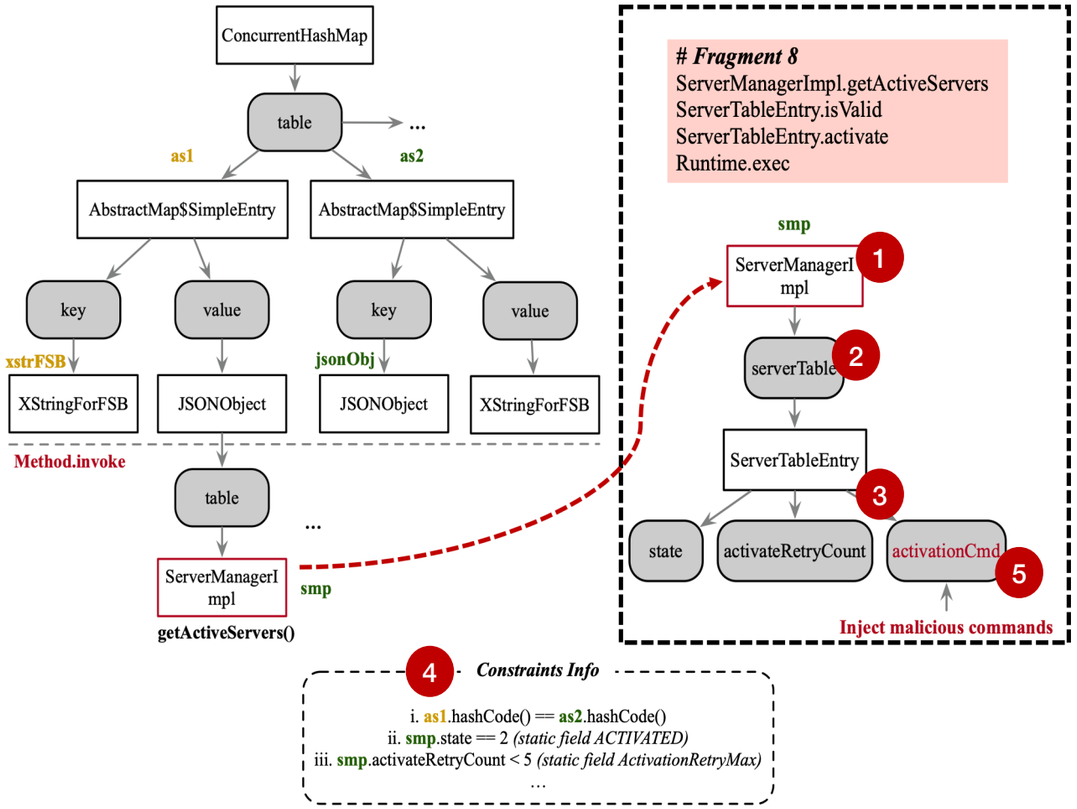


Illustration of IOCD

1. Object Initialization: Initializing different types of parameter-less Java instance objects based on the *Class-Node*.

2. Object Structure Recovery: 1) Establish the class hierarchy of these instances according to *directed edges*. 2) Set the *fields* related to attack *payload construction*.

3. Dominator Constraints Configuration: Extract *dominator constraints* and invoke the constraint solver to generate appropriate values, which are then assigned to the corresponding fields.

STEP5: IOCD-enhanced Directed Fuzzing

❖ Dependency-aware Mutation

Mutation Strategy

Property(*prop*) Type	Constraint(*cnst*) Type	Example	Mutation Strategy
Class	Class	if(*prop*==*cnst*)	set *prop* to *cnst*
	Method	if(*prop*.getDeclaredMethods().contains(*cnst*))	set *prop* to a Class instance that contains *cnst* Method
	String	if(*prop.name*==*cnst*)	set *prop* to a Class instance of *cnst*
		if(*prop.superClassName*==*cnst*)	set *prop* to a Sub-Class instance of *cnst*
		if(*prop.interfaceName*==*cnst*)	set *prop* to a Implementation Class instance of *cnst*
Method	Class	if(*prop.declaringClass*==*cnst*)	set *prop* to a Method instance of *cnst* Class
	String	if(*prop.name*.startWith(*cnst*))	set *prop* to a Method instance with proper name
	int	if(*prop.parameterNums*==0)	set *prop* to a Method instance with proper args numbers
Object	Class	if(*prop* instanceof *cnst*)	correct *prop* to an instance of *cnst*
Collection/Map	int	if(*prop*.size ≥ *cnst*)	add/remove elements
	Class	if(*prop.item* instanceof *cnst*)	correct the element type
	Object	if(prop.contains(cnst))	add/remove elements

Runtime Feedback-based Mutation

- JDD collects ***runtime feedback (e.g., covered branches, thrown exceptions)*** and uses this information to set corresponding fields accordingly.

Fixed Structure Mutation

- Based on IOCD, JDD ***fixes the structure of the Injection Object*** and only ***mutates fields that do not affect the overall structure***.

Field Dependency Mutation

- JDD considers the ***dependency relationships between fields*** to ensure that these dependencies are preserved during the mutation process

STEP5: IOCD-enhanced Directed Fuzzing

❖ Sink Reachable and Exploitable Verification

Sink Reachable Verification

- JDD ***instruments Sink methods*** such as `Runtime.exec` to help determining whether a Sink point has been reached.

Sink Exploitable Verification

- For regular Sinks (e.g., `Runtime.exec`), JDD ***directly injects malicious payloads into the relevant fields of the Injection Object***.
- For reflection-based Sinks (e.g., Method.invoke), JDD continues to ***search for related Gadget Fragments and links them to the existing Gadget chain***.

Evaluation and New Findings

Open-source repos: https://github.com/fdu-sec/JDD
https://github.com/BofeiC/JDD-PocLearning

Evaluation

Table 3: Gadget chain detection comparison among GadgetInspector, SerHybrid, ODDFuzz, and JDD (Our Approach). The number in parentheses indicates the detected known chains in the benchmark.

Application	Known Chains	GadgetInspector		SerHybrid		ODDFuzz		JDD		
		Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Identified Chains	Confirmed Chains	Unkown Chains
Ysoserial Benchmark										
AspectJWeaver	1	8	0	N/A	N/A	9	0			
CommonsBeantuils	1	4	0	0	0	8	1	108†	25	20
CommonsCollections	5	4	1	1	1	97	3			
BeanShell	1	2	0	1	0	8	0	587	5	4
C3P0	1	2	0	N/A	N/A	13	1	15	0	0
Click	1	4	0	N/A	N/A	8	1	6	1	0
Clojure	1	12	1	N/A	N/A	184	1	6	3	2
CommonsCollections4	2	4	0	1	1	112	2	230	26	24
Groovy	1	4	0	3	0	13	0	413	5	4
JavassistWeld	1	2	0	N/A	N/A	8	0	6	1	0
JBossInterceptors	1	2	0	N/A	N/A	8	0	7	1	0
JDK	4	5	0	N/A	N/A	9	1	16	8	5
JSON	1	7	0	N/A	N/A	9	0	147	6	5
Jython	1	42	1	N/A	N/A	32	0	0	0	0
MozillaRhino	2	3	0	N/A	N/A	7	2	4	2	0
Hibernate	2	3	0	3	0	8	2	14	5	4
Myfaces	2	2	0	N/A	N/A	7	0	52	3	2
ROME	1	2	0	0	0	5	1	48	9	8
Spring	2	2	0	N/A	N/A	10	0	5	0	0
Vaadin	1	6	0	N/A	N/A	13	1	109	14	13
FileUpload	1	3	0	N/A	N/A	8	0	1	1	0
Wicket	1	3	0	N/A	N/A	7	0	1	1	0
Total	34	126	3 (3)	9	2 (2)	583	16 (16)	1362	116 (27)	91
Recently Disclosed Vulnerabilities										
Weblogic	21	53	0	N/A	N/A	N/A	N/A	642	126	107
MarshalSec(Hessian)‡	5	2	0	N/A	N/A	N/A	N/A	119	38	33
Total	26	55	0 (0)	-	-	-	-	761	164 (24)	140

† Due to the shared use of the CommonsCollections dependency in detecting Gadget Chains in AspectJWeaver and CommonsBeanutils, we simultaneously conducted analysis on these three packages.
‡ MarshalSec is a deserialization vulnerability exploitation tool, and we include its Hessian protocol-based Exploits as part of our evaluation dataset.

Effectiveness

- ✓ JDD detects **91 unknown** gadget chains not detected by baselines
- ✓ JDD reduces the static false positive rate from 91.5% to 0% on Benchmark.

1362 (static detected)
→ 116 (dynamic verified)

Evaluation

JDD discovered **127 zero-day gadget chains** in six popular Java applications and notified affected developers to help them resolve the issues.

Application	Basic Information			Detected Chains			Performance		
	Stars	Class Number	Method Number	Identified Chains	Confirmed Chains	Vendor Reply	Search and Link Gadget Chains	Construct IOCD	Dynamic Verify (Detected Chains)
Apache Dubbo	39K	88.5K	936.4K	31	7	CVE Assigned	8min29s	53s	36min15s (31)
Motan	6K	53.7K	454.7K	695	93	CVE Assigned	15min25s	47min13s	6h (358)
Solon	1.5K	280.9K	2,797.9K	117	35	CVE Assigned	39min56s	29min16s	2h35min59s (117)
XXL-Job	24.5k	52.5K	411.1K	843	110	CVE Assigning	7min24s	52min8s	6h (363)
Sofa-rpc	4.8K	94.9K	883.9K	205	43	CVE Assigned	37min15s	8min6s	3h43min3s (205)
Apache Tapestry	0.1K	28.8K	241.1K	16	5	CVE Assigning	54s	9s	19min38s (16)

The Detected Chains and Performance Evaluation Results of JDD on Real-World Java Apps.

CVE-2023-29234	CVE-2023-35839	CVE-2023-39131
CVE-2023-48967	CVE-2024-23636	CVE-2023-41331

Assigned CVEs

New Findings - Gadget Chains

Known Gadget Chain

AnnotationInvocationHandler.readObject
Proxy Map.entrySet
ConversionHandler.invoke
ConvertedClosure.invokeCustom
Closure.call

Unknown Gadget Chain

ConcurrentHashMap.readObject
GString.hashCode
GString.toString
GString.writeTo
Closure.call

HashMap.put
HashMap.putVal
AbstractMap$SimpleEntry.equals
java.util.AbstractMap.access$000
java.util.AbstractMap.eq
XStringForFSB.equals
QBindingEnumeration.toString
ContextImpl.lookup

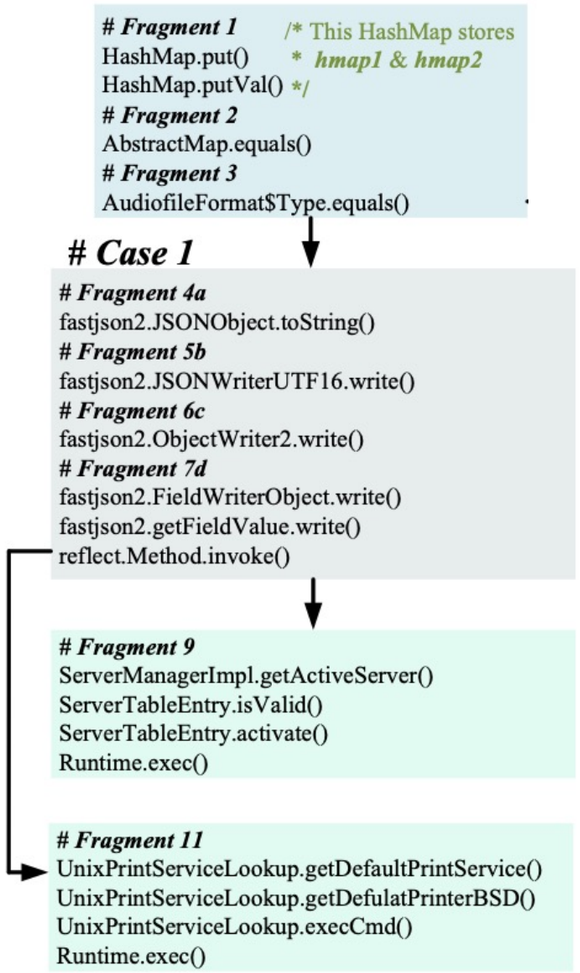
Expanding the range of protocols that can be attacked: the unknow gadget chain can be used to attack protocols that do not support dynamic proxy features, e.g. Hessian.

1

Can be used to attack a new protocol: the unknow gadget chain can be used to attack protocols outside the scope of JDD's predefined detection rules, e.g. Apache Fury.

2

New Findings - Gadget Chains

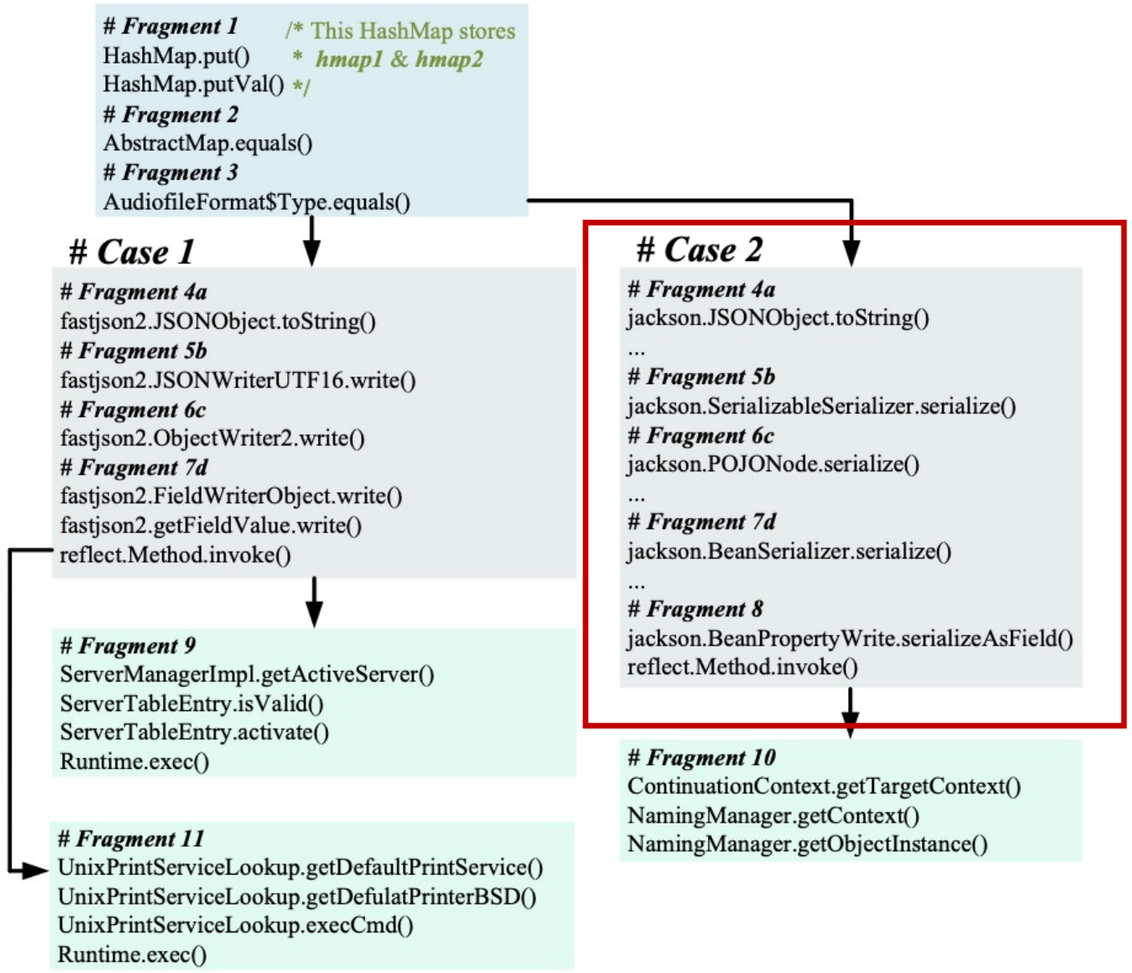


Zero-day Exploitable Gadget Chains

- **Case #1**
  - The exploitable gadget chain relies only on the JDK and a popular library (i.e., fastjson2).
  - Impacts many popular Java apps, e.g. Sofa, Solon…

However, it cannot be used to exploit Motan (fastjson is not introduced by default)

New Findings - Gadget Chains



- **Case #2**
  - An evolution of Case #1.

We found that by replacing certain fragments 4a–7d in Case #1, we can generate new gadget chains to exploit other apps, e.g., Motan(weibo).

Zero-day Exploitable Gadget Chains

Gadget fragments with high reuse value

> => `Comparator.compare`

Known:

java.util.PriorityQueue: void readObject(java.io.ObjectInputStream)
  java.util.PriorityQueue: void heapify()
    java.util.PriorityQueue: void siftDown(int,java.lang.Object)
      java.util.PriorityQueue: void siftDownUsingComparator(int,java.lang.Object)
        java.util.Comparator: int compare(T o1, T o2)

JDD discovered:

java.util.concurrent.ConcurrentHashMap: boolean equals(java.lang.Object)
  java.util.concurrent.ConcurrentSkipListMap: java.lang.Object get(java.lang.Object)
    java.util.concurrent.ConcurrentSkipListMap: java.lang.Object doGet(java.lang.Object)
      java...ConcurrentSkipListMap: int cpr(java.util.Comparator,...Object,...Object)
        java.util.Comparator: int compare(T o1, T o2)

Gadget fragments with high reuse value

> `Object.equals`(Can satisfy the hash collision condition) => `Object.equals`

org.springframework.aop.target.HotSwappableTargetSource: boolean equals(java.lang.Object)
    java.lang.Object: boolean equals(java.lang.Object)

```
java.util.AbstractMap$SimpleEntry: boolean equals(java.lang.Object)
    java.util.AbstractMap: boolean access$000(java.lang.Object,java.lang.Object)
        java.util.AbstractMap: boolean eq(java.lang.Object,java.lang.Object)
            java.lang.Object: boolean equals(java.lang.Object)

java.util.AbstractMap$SimpleImmutableEntry: boolean equals(java.lang.Object)
    java.util.AbstractMap: boolean access$000(java.lang.Object,java.lang.Object)
        java.util.AbstractMap: boolean eq(java.lang.Object,java.lang.Object)
            java.lang.Object: boolean equals(java.lang.Object)
```

Gadget fragments with high reuse value

> => `Object.toString`

com.sun.org.apache.xpath.internal.objects.XString: boolean equals(java.lang.Object)
  java.lang.Object: java.lang.String toString()

com.sun.org.apache.xpath.internal.objects.XStringForFSB: boolean equals(java.lang.Object)
    java.lang.Object: java.lang.String toString()

javax.sound.sampled.AudioFormat$Encoding: boolean equals(java.lang.Object)
    java.lang.Object: java.lang.String toString()

javax.sound.sampled.AudioFileFormat$Type: boolean equals(java.lang.Object)
    java.lang.Object: java.lang.String toString()

javax.swing.UIDefaults$TextAndMnemonicHashMap: java.lang.Object get(java.lang.Object)
    java.lang.Object: java.lang.String toString()

Gadget fragments with high reuse value

> getter => Command/JNDI injection attack E.g. Linked after unsafe reflection

```
com.sun.corba.se.impl.activation.ServerManagerImpl: int[] getActiveServers()
    com.sun.corba.se.impl.activation.ServerTableEntry: boolean isValid()
        com.sun.corba.se.impl.activation.ServerTableEntry: void activate()
            java.lang.Runtime: java.lang.Process exec(java.lang.String)

com.p6spy.engine.spy.P6DataSource: java.sql.Connection getConnection()
    com.p6spy.engine.spy.P6DataSource: void bindDataSource()
        javax.naming.InitialContext: java.lang.Object lookup(java.lang.String)

com.zaxxer.hikari.hibernate.HikariConnectionProvider: java.sql.Connection getConnection()
    com.zaxxer.Hikari.HikariDataSource: java.sql.Connection getConnection()
        com.zaxxer.hikari.pool.HikariPool: HikariPool(com.zaxxer.Hikari.HikariConfig)
            com.zaxxer.hikari.pool.PoolBase: PoolBase(com.zaxxer.Hikari.HikariConfig)
                com.zaxxer.hikari.pool.PoolBase : void initializeDataSource()
                    javax.naming.InitialContext: java.lang.Object lookup(java.lang.String)
```

Conclusion & Takeaways

- We introduced a fragment-based summary and a bottom-up gadget chain search approach that effectively addresses the challenge of static path explosion.
- JDD uses a technical framework that leverages a lightweight static taint analysis engine to guide directed fuzzing, thereby enhancing precision and efficiency in vulnerability verification.
- We also shared several zero-day exploitable gadget chains and fragments.