

# Reverse Engineering 101

Dozent: Prof. Dr. Michael Eichberg  
Kontakt: michael.eichberg@dhbw.de  
Version: 1.0.1

---

Folien: <https://delors.github.io/sec-reversing101/folien.de.rst.html>  
<https://delors.github.io/sec-reversing101/folien.de.rst.html.pdf>  
Fehler melden: <https://github.com/Delors/delors.github.io/issues>

# Vorerfahrungen?

- Wer hat schon einmal Software or Hardware Reverse Engineering betrieben?
- Wer kennt Java Bytecode?
- Wer hat Erfahrung mit Python?

# Reverse Engineering

Reverse Engineering ist die Analyse von Systemen mit dem Ziel, ihren Aufbau und ihre Funktionsweise zu verstehen.

Typische Anwendungsfälle:

- die Rekonstruktion (von Teilen) des Quellcodes von Programmen, die nur als Binärabbild vorliegen.
- die Analyse von Kommunikationsprotokollen proprietärer Software

---

Vom Reverse Engineering ist das **Reengineering** zu unterscheiden. Im Fall von letzteren geht es „nur“ darum die Funktionalität eines bestehenden Systems mit neuen Techniken wiederherzustellen.

# Zweck von Reverse Engineering

- Herstellung von Interoperabilität
- Untersuchung auf Schwachstellen
- Untersuchung auf Copyrightverletzungen
- Untersuchung auf Backdoors
- Analyse von Viren, Würmern etc.
- Umgehung von ungerechtfertigten(?) Schutzmaßnahmen (z. B. bei Malware)

## ***CPU-Lücke macht Malware-Infektionen nahezu unumkehrbar***

*Die Schwachstelle verschafft Angreifern Zugang zu einer der höchsten Privilegienstufen heutiger PC-Systeme. Schadsoftware entzieht sich damit jeglicher Erkennung.*

*[...] Gegenüber Wired erklärten die Forscher, per Sinkclose ließen sich etwa Bootkits installieren, die für das Betriebssystem und gängige Antivirensoftware unsichtbar seien, während Angreifer einen Vollzugriff auf das Zielsystem erhielten.*

*—August, 2024 - **Golem.de** (AMD CVE)*

# CVE-2024-3094 - liblzma Backdoor in OpenSSH[1][2]

Ziel	Wie verbreitet?	Bewertung
<p>Das Verhalten von SSH bei der Authentifikation so zu verändern, dass es dem Angreifer Zugang zum System erlaubt.</p> <p>Zur Absicherung der Backdoor ist diese über ein Zertifikat abgesichert.</p>	<p>Die Bibliothek <code>liblzma</code> wurde so angepasst, dass diese eine Backdoor in SSH einbaut.</p> <p>Der Schadcode ist nur in den Tarballs zu finden - nicht im SourceCode im GIT. Der eigentliche Schadcode wurde versteckt in <i>Testfixtures</i>.</p> <p>Der Code wurde so entworfen, dass bekannte Werkzeuge (<i>Valgrind</i>) keine Probleme erkennen.</p> <p>Die Bibliothek wurde nur in bestimmten Situationen von OpenSSH verwendet.</p>	<p>CVSS Base Score: 10.0 (kritisch)</p> <p><i>Entstandener Schaden:</i> vermutlich gering, da (gerade noch) keine offiziellen Releases (von Debian, Ubuntu, etc.) betroffen waren.</p> <p>Dem Angriff ging ein sehr langer Social Engineering Angriff voraus, weswegen mit höherer Wahrscheinlichkeit ein „State-sponsored Actor“ dahintersteckt.</p>

[1] InnoQ Podcast


[2] SSH Blob

## Backdoor in 16 D-Link Routern[3]

- Angreifer können aus dem lokalen Netzwerk heraus den Telnet-Dienst betroffener D-Link-Router durch Angabe einer bestimmten Ziel URL aktivieren.
  - Die Admin-Zugangsdaten sind in der Firmware hinterlegt.
  - Vermutlich ursprünglich für werksseitige Tests.
  - *CVSS Base Score*: 8.8 (hoch)
- 

[3] [Golem.de](#)

# Reverse Engineering - grundlegende Schritte

- 01** Informationsgewinnung zur Gewinnung aller relevanten Informationen über das Produkt.
- 02** Modellierung mit dem Ziel der (Wieder-)Gewinnung eines (abstrakten) Modells der relevanten Funktionalität.
- 03** Überprüfung ( *review*) des Modells auf seine Richtigkeit und Vollständigkeit.



# Informationsgewinnung - Beispiel

Gegeben sei eine App zum Ver- und Entschlüsseln von Dateien sowie ein paar verschlüsselte Dateien. Mögliche erste Schritte vor der Analyse von Binärcode:

- Die ausführbare Datei ggf. mit `file` (oder sogar mit `binwalk`) überprüfen (z. B. wie wurde die Datei kompiliert und für welches Betriebssystem und Architektur)

Beispiel:

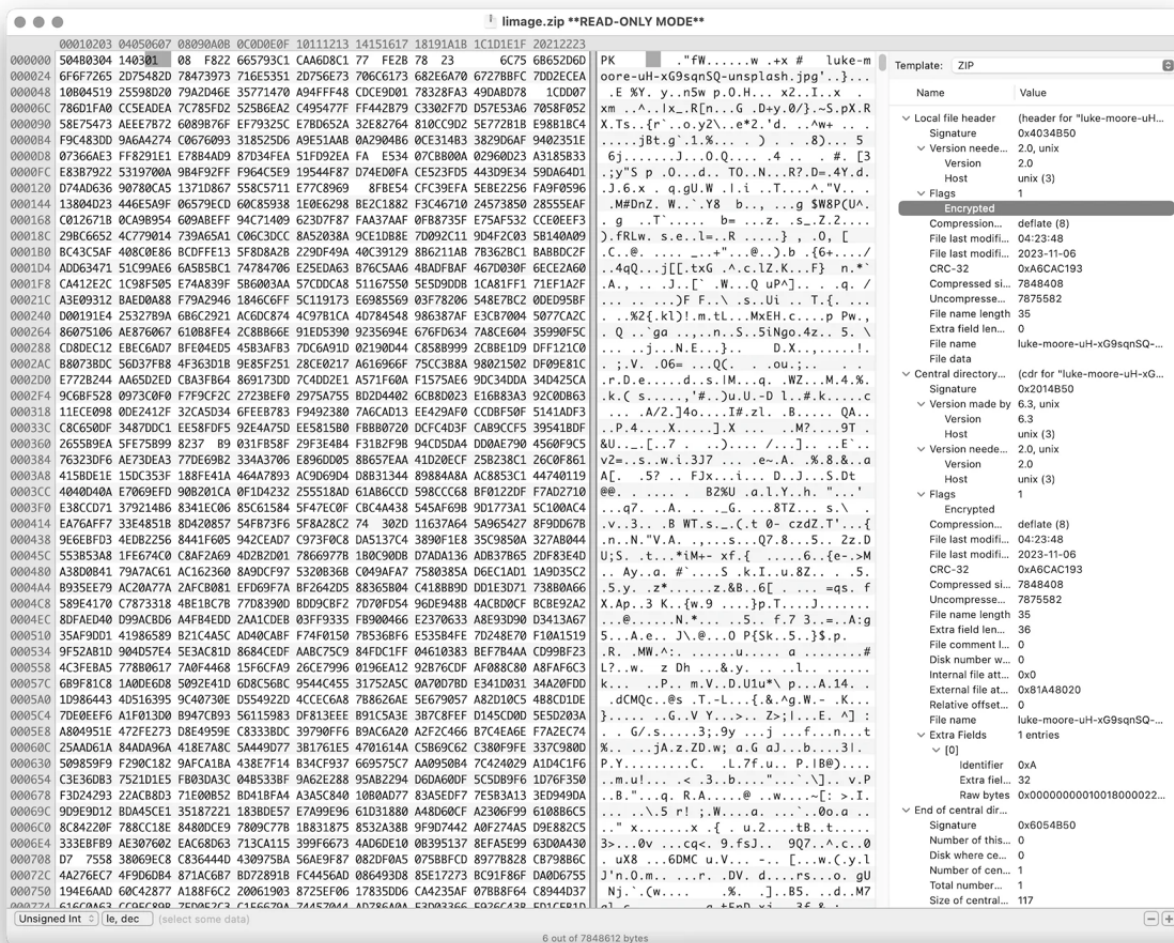
```
$ file /usr/bin/openssl
```

```
/usr/bin/openssl: Mach-O universal binary with 2 archi...
```

```
/usr/bin/openssl (for architecture x86_64): Mach-O 64-bit
```

```
/usr/bin/openssl (for architecture arm64e): Mach-O 64-bit
```

- Die Dateien mit einem (guten) Hexeditor auf Auffälligkeiten untersuchen.



Vorsicht!

Die Datei ggf. auf (bekannte) Viren und Malware überprüfen.

- Eine Datei mit einem bekannten Inhalt verschlüsseln und danach vergleichen.

Ist die Datei gleich groß?

Falls ja, dann werden keine Metainformationen gespeichert und das Passwort

kann (ggf.) nicht (leicht) verifiziert werden.

(Es kann zumindest nicht *direkt* in der Datei gespeichert sein.)

- Eine Datei mit verschiedenen Passworten verschlüsseln.

Sind die Dateien gleich?

Falls ja, dann wäre die Verschlüsselung komplett nutzlos und es gilt nur noch den konstanten Schlüssel zu finden.

Gibt es Gemeinsamkeiten?

Falls ja, dann wäre es möglich, dass das Passwort (gehasht) in der Datei gespeichert wird.

- Eine Datei mit einem wohldefinierten Muster verschlüsseln, um ggf. den „Mode of Operation“ (insbesondere ECB) zu identifizieren.
- Mehrere verschiedene Dateien mit dem gleichen Passwort verschlüsseln

Gibt es Gemeinsamkeiten?

Falls ja, dann wäre es möglich, dass die entsprechenden Teile direkt vom Passwort abgeleitet werden/damit verschlüsselt werden.

- ...

- Reverse Engineering der App durchführen.

# Rechtliche Aspekte des Reverse Engineering

## Vorsicht!

Die Gesetzgebungen unterscheiden sich von Land zu Land teils signifikant.

- Die Rechtslage hat sich in Deutschland mehrfach geändert.
- Umgehung von Kopierschutzmechanismen ist im Allgemeinen verboten.
- Lizenzen verbieten das Reverse Engineering häufig; es stellt sich aber die Frage nach der Rechtmäßigkeit der Klauseln.

## Warnung

Bevor Sie Reverse Engineering von Systemen betreiben, erkundigen sie sich erst über mögliche rechtliche Konsequenzen.

# 1. Software Reverse Engineering

---

# Ansätze

**statische Analyse:** Studieren des Programms ohne es auszuführen; typischerweise mittels eines Disassemblers oder eines Decompilers.

**dynamische Analyse:**

Ausführen des Programms; typischerweise unter Verwendung eines Debuggers oder eines instrumentations Frameworks (z. B. **Frida**).

**hybride Analyse:** Kombination aus statischer und dynamischer Analyse.

Ansätze wie **Unicorn**, welches auf **QEmu** aufbaut, erlaubt zum Beispiel die Ausführung von (Teilen von) Binärcode auf einer anderen Architektur als der des Hosts.

Ein Beispiel wäre die Ausführung einer Methode, die im Code verschlüsselte hinterlegte Strings entschlüsselt (🚧 *deobfuscation*), um die Analyse zu vereinfachen.

Ggf. müssen für Teile des Codes, die die Hostfunktionalität nutzen, Stubs/Mocks bereitgestellt werden.

# Disassembler

Überführt (maschinenlesbaren) Binärcode in Assemblercode

Kommandozeilenwerkzeuge (exemplarisch):

- `objdump -d`
- `gdb`
- `radare`
- `javap` (für Java)

## Hinweis

Für einfache Programme ist es häufig möglich direkt den gesamten Assemblercode mittels der entsprechenden Werkzeuge zu erhalten. Im Falle komplexer Binärdateien (z. B. im ELF (Linux) und PE (Windows) Format) gilt dies nicht und erfordert ggf. manuelle Unterstützung zum Beispiel durch das Markieren von Methodenanfängen. Im Fall von Java `.class` ist die Disassemblierung immer möglich.

# Decompiler

Überführt (maschinenlesbarem) Binärcode *bestmöglich* in Hochsprache (meist C ähnlich oder Java). Eine *kleine* Auswahl von verfügbaren Werkzeugen:

- Hex-Rays IDAPro (kommerziell)
- Ghidra (unterstützt fast jede Plattform; die Ergebnisse sind sehr unterschiedlich.)
- JadX (Androids .dex Format)
- CFR (Java .class Dateien)
- IntelliJ

---

Mittels Decompiler ist es ggf. möglich Code, der zum Beispiel ursprünglich in Kotlin oder Scala geschrieben und für die JVM kompiliert wurde, als Java Code zurückzubekommen.

Die Ergebnisse sind für Analysezwecke zwar häufig ausreichend gut – von funktionierendem Code jedoch ggf. (sehr) weit entfernt.

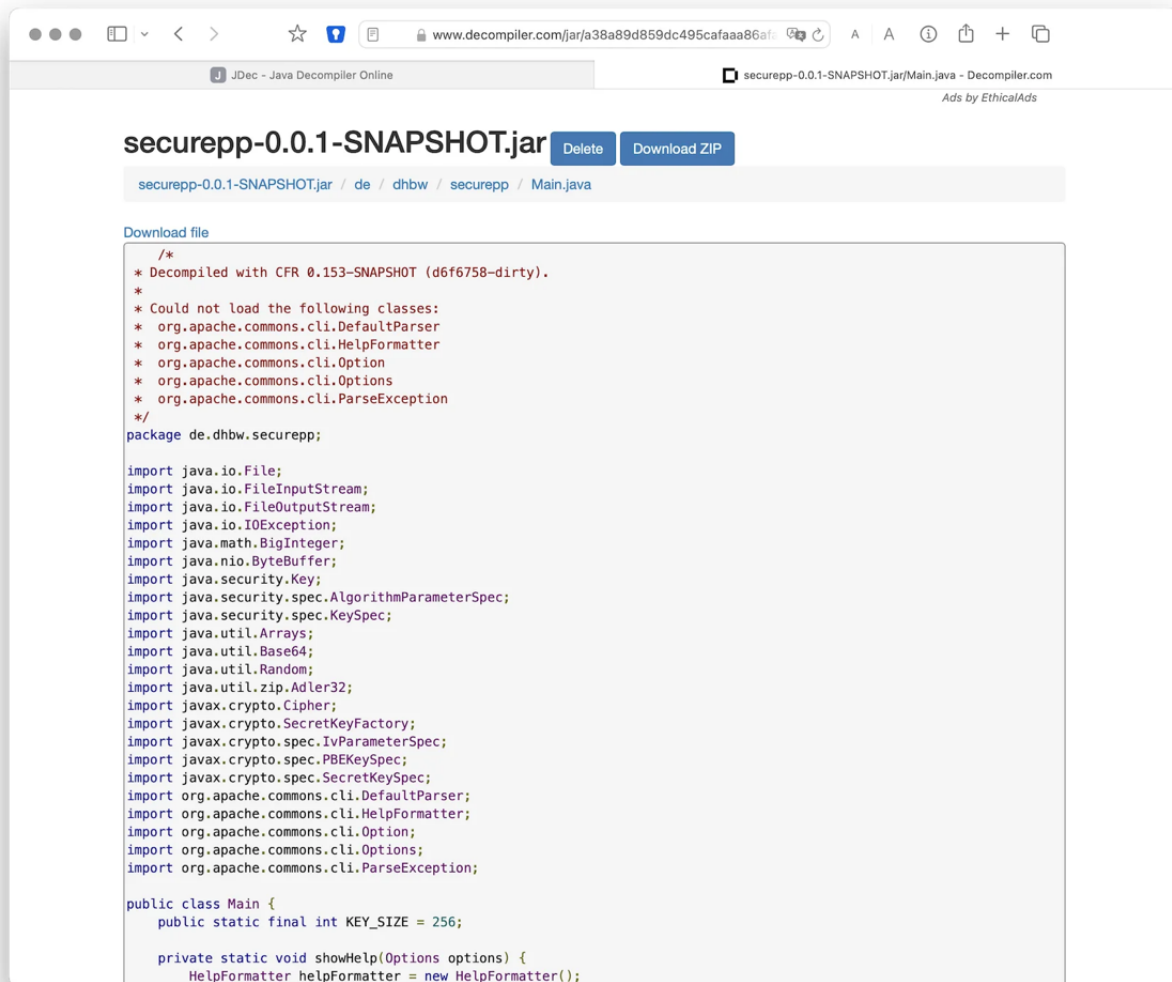
[decompiler.com](https://decompiler.com) unterstützt eine große Anzahl ausführbaren Dateien.

## Hinweis

Decompiler sind generell sehr hilfreich, aber gleichzeitig auch sehr fehlerbehaftet. Vieles, was im Binärcode möglich ist, hat auf der Ebene des Sourcecodes keine Entsprechung.

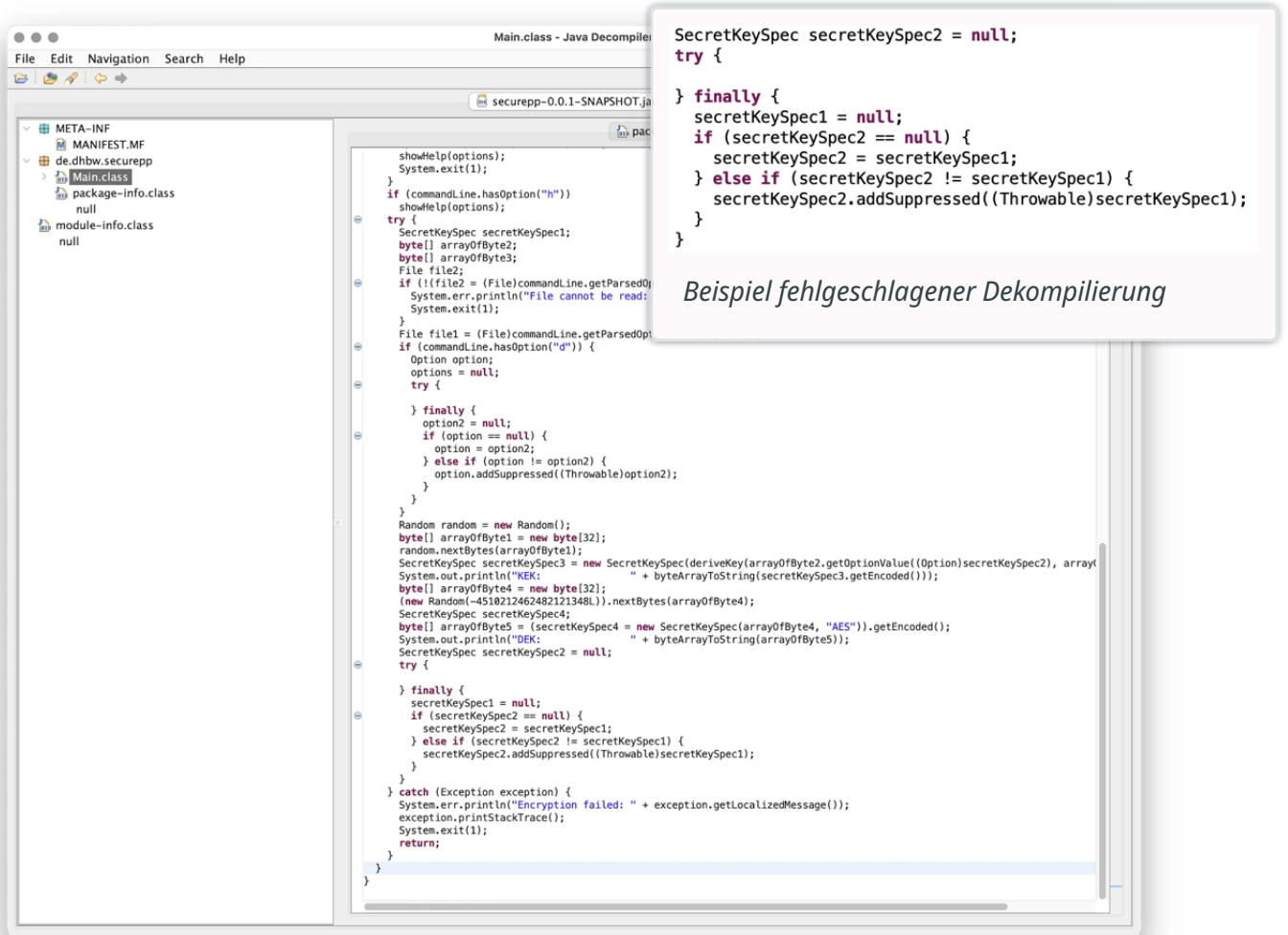
Zum Beispiel unterstützt Java Bytecode beliebige Sprünge. Solche Code wird aber durch normale Programme, die z. B. in Java, Kotlin, Scala oder Clojure geschrieben wurden, nicht erzeugt. Decompiler kommen mit solchem Code in der Regel nicht (gut) zurecht.

# cfr Decompiler





# JD Decompiler

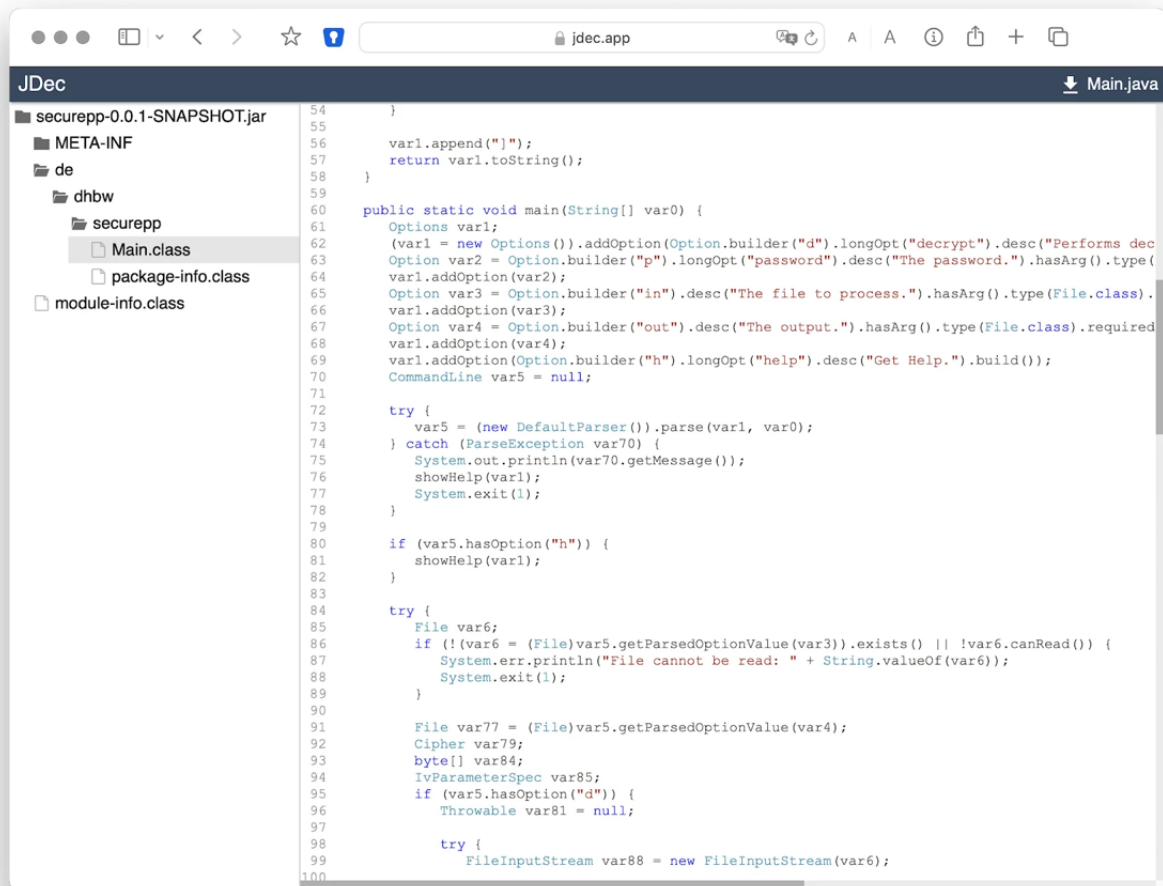


The screenshot shows the JD Decompiler interface. On the left, a project tree for 'securepp-0.0.1-SNAPSHOT.jar' is visible, containing 'META-INF', 'MANIFEST.MF', 'de.dhbw.securepp', 'Main.class', 'package-info.class', and 'module-info.class'. The main window displays the decompiled code for 'Main.class'. The code includes a 'showHelp' method, a 'main' method with command-line argument parsing, and a 'try-finally' block for handling 'SecretKeySpec' objects. A callout box highlights a specific code block with the text 'Beispiel fehlgeschlagener Dekompilierung' (Example of failed decompilation).

```
SecretKeySpec secretKeySpec2 = null;
try {
} finally {
    secretKeySpec1 = null;
    if (secretKeySpec2 == null) {
        secretKeySpec2 = secretKeySpec1;
    } else if (secretKeySpec2 != secretKeySpec1) {
        secretKeySpec2.addSuppressed((Throwable)secretKeySpec1);
    }
}
```

*Beispiel fehlgeschlagener Dekompilierung*

# JDec Decompiler



# Debugger

Dient der schrittweisen Ausführung des zu analysierenden Codes oder Hardware; ermöglichen zum Beispiel Speicherinspektion und Manipulation.

- gdb
- lldb
- x64dbg (Windows, Open-Source)
- jdb (Java Debugger)

---

## Hardware Debugger

Für das Debuggen von Hardware gibt es entsprechende Werkzeuge, z. B. **Lauterbach Hardware Debugger** (kommerziell und sehr teuer).

Mittels solcher Werkzeuge ist es möglich die Ausführung von Hardware Schritt für Schritt (🖱️ *single step mode*) zu verfolgen und den Zustand der Hardware (Speicher und Register) zu inspizieren. Dies erfordert jedoch häufig eine JTAG Schnittstelle oder etwas vergleichbares.

## 2. Erschwerung des Reverse Engineering



# Obfuscation → für Menschen unverständlich Code

- Techniken, die dazu dienen das Reverse Engineering zu erschweren.
- Häufig eingesetzt ...
- von Malware
- Adware (im Kontext von Android ein häufig beobachtetes Phänomen)
- zum Schutz geistigen Eigentums
- für DRM / Durchsetzung von Kopierrechten
- zur Prävention von „Cheating“ (insbesondere im Umfeld von Online Games)
- Wenn das Programm als Source Code vertrieben wird bzw. vertrieben werden muss (JavaScript)
- Arbeiten auf Quellcode oder Maschinencode Ebene
- Grenze zwischen *Code Minimization*, *Code Optimization* und *Code Obfuscation* ist fließend.
- Mögliche Werkzeuge (ohne Wertung der Qualität/Effektivität):
- [Java] Proguard / Dexguard
- [C/C++] **Star Force**

---

Gerade im Umfeld von klassischen *Binaries* für Windows, Mac und Linux erhöhen Compiler Optimierungen, z. B. von C/C++ und Rust Compilern (`-O2` / `-O3`), bereits den Aufwand, der notwendig ist den Code zu verstehen, erheblich.

## Hinweis

Einen ambitionierten und entsprechend ausgestatteten Angreifer wird **Code Obfuscation** bremsen, aber sicher nicht vollständig ausbremsen und das Vorhaben verteilen.

# Obfuscation - Techniken (Auszug)

- entfernen aller Debug-Informationen
- Das Kürzen aller möglichen Namen (insbesondere Methoden und Klassennamen).
- Das Verschleiern von Konstanten durch den Einsatz vermeintlich komplexer Berechnungen zu deren Initialisierung.

```
~(((int)Math.PI) ^ Integer.MAX_VALUE >> 16)+Short.MAX_VALUE  
= 2
```

- Die Verwendung von Unicode Codepoints für Strings oder die Verschleierung von Strings mittels **rot13** Verschlüsselung.

```
/* ??? */ printf("\x48""e\x154l\x6F"" \x127o\x72""l\x144!");  
/* = */ printf("Hello World!");
```

- Das Umstellen von Instruktionen, um das Dekompilieren zu erschweren.
- Das Hinzufügen von totem Code.
- Den relevanten Teil der Anwendung komprimieren und verschlüsseln und erst bei Verwendung entpacken und entschlüsseln.
- ...

---

Obfuscation auf Source Code Ebene: **International Obfuscated C Code Contest**

## Umstellen von Instruktionen

Das Umstellen von Instruktionen erschwert die Analyse, da viele Werkzeuge zum Dekompilieren auf die Erkennung von bestimmten Mustern im Code angewiesen sind und ansonsten nur sehr generischen (Spagetti Code) oder gar unsinnigen Code zurückgeben.

## Verschleierung von Strings

Das Verschleiern von Strings kann insbesondere das Reversen von Binärcode erschweren, da ein Angreifer häufig „nur“ an einer ganz bestimmten Funktionalität interessiert ist und dann Strings ggf. einen sehr guten Einstiegspunkt für die weitergehende Analyse bieten.

Stellen Sie sich eine komplexe Java Anwendung vor, in der alle Namen von Klassen, Methoden und Attributen durch einzelne oder kurze Sequenzen von Buchstaben ersetzt wurden und sie suchen danach wie von der Anwendung Passworte verarbeitet werden. Handelt es sich um eine GUI Anwendung, dann wäre zum Beispiel die Suche nach Text, der in den Dialogen vorkommt (z. B. "Password") z. B. ein sehr guter Einstiegspunkt.





# ClassLoader

- `ClassLoader` dienen dazu Klassen dynamisch zu laden. D. h. eine Klasse wird erst dann von der JVM geladen, wenn sie benötigt bzw. angefordert wird.
- Jeder `ClassLoader` spannt seinen eigenen Namensraum auf.  
Zwei Instanzen der gleichen Klasse (d. h. mit dem selben Bytecode) sind nicht gleich (Referenzgleichheit), wenn zwei verschiedene `ClassLoader` genutzt wurden.
- `ClassLoader` stehen in einer Hierarchie.
- `ClassLoader` können genutzt werden, um:
  - ein Programm dynamisch zu erweitern (Plug-ins)
  - um Klassen zu laden, die zur Laufzeit generiert wurden
  - um den Bytecode zu manipulieren, bevor er von der JVM ausgeführt wird.

# Ein eigener ClassLoader

```
1 static class MyClassLoader extends ClassLoader {
2     public MyClassLoader(ClassLoader parent) { super(parent); }
3
4     @Override
5     protected Class<?> findClass(final String name) throws ClassNotFoundException {
6         try (final var in = super.getResourceAsStream(name)) {
7             final var classBytes = new byte[in.available()];
8             final var readBytes = in.read(classBytes);
9             if (readBytes != classBytes.length) {
10                 throw new IOException("failed reading class file: " + name);
11             }
12             return defineClass(name, classBytes, 0, classBytes.length);
13         } catch (IOException ioe) {
14             throw new ClassNotFoundException("failed loading " + name, ioe);
15         } } }
```

# ClassLoading - Example

## Ein Singleton

```
1 public class MySingleton {
2
3     private static MySingleton instance = null;
4     private MySingleton() {}
5
6     public static synchronized MySingleton instance() {
7         if (instance == null) instance = new MySingleton();
8         return instance;
9     }
10 }
```

## Gleichheit von Instanzen

```
1 Object a = MySingleton.instance();
2 Object b = MySingleton.instance();
3 System.out.println(a == b);
```

Ergebnis: true

## Verwendung des SystemClassLoader

```
1 ClassLoader cl1 = ClassLoader.getSystemClassLoader();
2 Class<?> clazz1 = cl1.loadClass("demo.MySingleton");
3 Object a = clazz1
4     .getDeclaredMethod("instance", new Class<?>[] { })
5     .invoke(null);
6 ClassLoader cl2 = ClassLoader.getSystemClassLoader();
7 Class<?> clazz2 = cl2.loadClass("demo.MySingleton");
8 Object b = clazz2
9     .getDeclaredMethod("instance", new Class<?>[] { })
10    .invoke(null);
11
12 System.out.println(a == b);
```

Ergebnis: true

## Verwendung von zwei Instanzen von MyClassLoader

```
1 ClassLoader cl1 = new MyClassLoader();
2 Class<?> clazz1 = cl1.loadClass("demo.MySingleton");
3 Object a = clazz1
4     .getDeclaredMethod("instance", new Class<?>[] { })
5     .invoke(null);
6 ClassLoader cl2 = new MyClassLoader();
7 Class<?> clazz2 = cl2.loadClass("demo.MySingleton");
8 Object b = clazz2
9     .getDeclaredMethod("instance", new Class<?>[] { })
10    .invoke(null);
11
12 System.out.println(a == b);
```

Ergebnis: false



### 3. Eine sehr kurz Einführung in Java Bytecode

# Die Java Virtual Machine



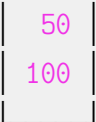

- **Java Bytecode** ist die Sprache, in der Java (oder Scala, Kotlin, ...) Programme auf der Java Virtual Machine (JVM) [4] ausgeführt werden.
  - In den meisten Fällen arbeiten Java Decompiler so gut, dass ein tiefgehendes Verständnis von Java Bytecode selten notwendig ist.
  - Java Bytecode kann — muss aber nicht — interpretiert werden. (Z. B. können „virtuelle Methodenaufrufe“ in Java schneller sein als in C++.)
- 

[4] [Java Bytecode Spezifikation](#)

# Java Bytecode - stackbasierte virtuelle Maschine

Die JVM ist eine stackbasierte virtuelle Maschine.

Die getypten Operanden eines Befehls werden auf einem Stack abgelegt und die Operationen arbeiten auf den obersten Elementen des Stacks. Jeder Thread hat seinen eigenen Stack.

Instruktionen	Veränderung des Stacks
<code>nop</code>	
<code>bipush 100</code> $\rightarrow$ <code>int</code>	
<code>bipush 50</code> $\rightarrow$ <code>int</code>	
<code>iadd</code> $\leftarrow 2 \times \text{int} \rightarrow \text{int}$	

-----

Eine Methode muss einen Stack begrenzter Höhe aufweisen. Code, für den die Stackhöhe nicht berechenbar ist, wird vom Compiler abgelehnt. (Zum Beispiel ein `bipush` in einer Endlosschleife.) Die benötigte Höhe des Stacks wird vom Compiler berechnet und von der JVM überprüft.

# Java Bytecode - Methodenaufrufe und lokale Variablen

- Die Java Virtual Machine verwendet lokale Variablen zur Übergabe von Parametern beim Methodenaufruf.
- Beim Aufruf von *Klassenmethoden* (**static**) werden alle Parameter in aufeinanderfolgenden lokalen Variablen übergeben, beginnend mit der lokalen Variable 0. D. h. in der aufrufenden Methode werden die Parameter vom Stack geholt und in lokalen Variablen gespeichert.
- Beim Aufruf von *Instanzmethoden* wird die lokale Variable 0 dazu verwendet, um die Referenz (**this**) auf das Objekt zu übergeben, auf dem die Instanzmethode aufgerufen wird. Anschließend werden alle Parameter in aufeinanderfolgenden lokalen Variablen übergeben, beginnend mit der lokalen Variable 1.

---

Die Anzahl der benötigten lokalen Variablen wird vom Compiler berechnet und von der JVM überprüft.



# Beispiel: *Default Constructor* In Java Bytecode

Ein *Constructor* welcher keine expliziten Parameter hat und nur den super Konstruktor aufruft.

```
1 // Method descriptor ()V
2 // Stack: 1, Locals: 1
3 public Main();
4     0  aload_0 [this]
5     1  invokespecial java.lang.Object()
6     4  return
```

Die Zeilennummern und die Informationen über die lokalen Variablen sind optional und werden nur für Debugging Zwecke benötigt.

```
Line numbers: [pc: 0, line: 9]
Local variable table: [pc: 0, pc: 5] local: this
                      index: 0
                      type:  de.dhbw.simplesecurepp.Main
```

---

Es gibt weitere Metainformationen, die „nur“ für Debugging-Zwecke benötigt werden, z. B. Informationen über die ursprüngliche Quelle des Codes oder die sogenannte "Local Variable Type Table" in Hinblick auf generische Typinformationen. Solche Informationen werden häufig vor Auslieferung entfernt bzw. nicht hineinkompiliert.

## Beispiel: Aufruf einer komplexeren Methode

```
1 // Method descriptor ([Ljava/lang/String;)V
2 // Stack: 5, Locals: 8
3 public static void main(java.lang.String[] args) throws ...;
4     0  aload_0 [args]
5     1  arraylength
6     2  iconst_2
7     3  if_icmpeq 74 // integer comparison for equality
8     6  getstatic java.lang.System.err : java.io.PrintStream
9     9  ldc <String "SimpleSecure++">
10    11  invokevirtual java.io.PrintStream.println(java.lang.String) : void
11    ...
```

## 4. Verschlüsselung von Daten

---

# Alternativen zur Speicherung von Passwörtern

In einigen Anwendungsgebieten ist es möglich auf das explizite Speichern von Passwörtern ganz zu verzichten[\*].

Stattdessen wird z. B. einfach versucht das Ziel zu entschlüsseln und danach evaluiert ob das Passwort (vermutlich) das Richtige war.

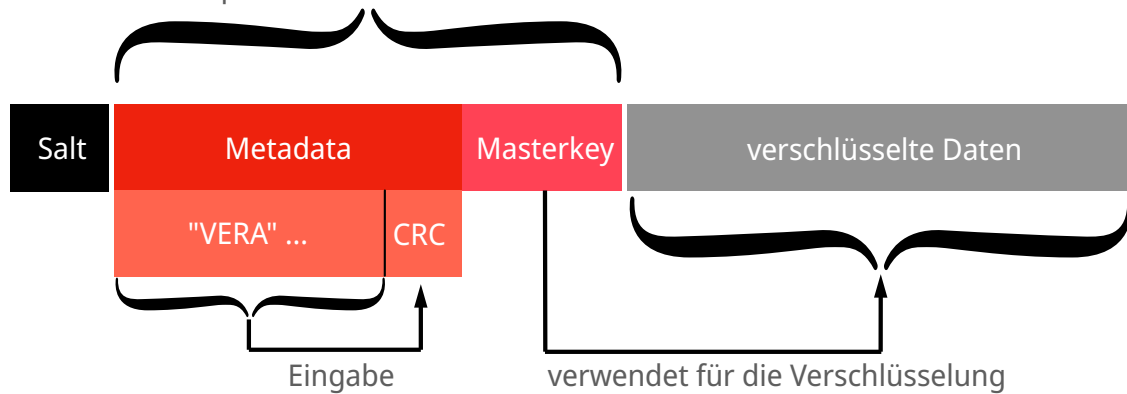
Kann darauf verzichtet werden zu überprüfen ob das Passwort korrekt war, dann sind keine Metainformationen notwendig und die verschlüsselte Datei kann genau so groß sein wie die unverschlüsselte Datei.

---

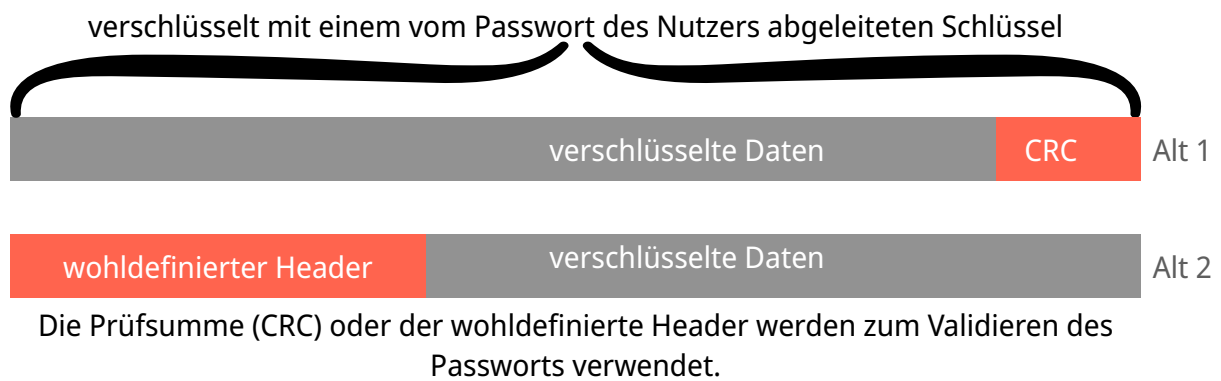
[\*] Bei einer Verschlüsselung mit OpenSSL wird das Passwort nicht gespeichert.

# Schematische Darstellung der Verschlüsselung von Containern (z. B. Veracrypt)

Der Schlüssel wird mit Hilfe einer Schlüsselableitungsfunktionen aus dem Nutzerpasswort berechnet.



# Generische Dateiverschlüsselung ohne explizite Speicherung des Passworts



**!! Wichtig**

Bleibe fokussiert!

Analysiere nur was notwendig ist.

# Live Exercise

Gegeben

Programm: Simplesecure++

Datei: 42.enc

Hinweise: Hints.pdf

## 4.1. Reversing SimpleSecure++



# Reverse Engineering Übung

Gegeben

Programm:

Secure++

**Exemplarische Verwendung zum Verschlüsseln**

```
java -jar securepp-0.0.1.jar de.dhbw.securepp.Main \  
    -p 'VielleichtIstEsRichtig-vielleichtAuchNICHT...' \  
    -in Poem.txt -out Poem.enc
```

Datei:

Poem.enc

Hinweise:

Hints.pdf

## 4.2. Reversing Secure++

Entschlüsseln Sie die Datei Poem.enc, die mit Secure++ verschlüsselt wurde.