

Java Security im Überblick

EUROSEC GmbH Chiffriertechnik & Sicherheit

Tel: 06173 / 60850, www.eurosec.com

© EUROSEC GmbH Chiffriertechnik & Sicherheit, 2005

Überblick

- Teil 1: Java Programmieretechniken zur Erhöhung der Sicherheit
- Teil 2: Schutz von Java-Source-Code vor Reverse Engineering

Java und Sicherheit

- Java gilt als „sichere Programmiersprache“
 - Keine Buffer Overflow Problematik
 - Memory Management durch VM
 - Integrierte Sicherheitsprüfungen
 - Byte-Code Checks
 - Klassen/Methoden Sichtbarkeit
 - Sicherheitseigenschaften verhindern jedoch keine Probleme durch Programmier-“Fehler“
 - Woran muss man denken? Wichtig sind Empfehlungen
 - zu grundlegenden Programmier-Techniken, und
 - zum Erhöhen der Sicherheit
- Best Practice Programmier-Techniken erforderlich

Basis-Programmiertechniken (1)

- Verwendung von public Klassenvariablen einschränken
 - Als privat definieren
 - Entsprechende Zugriffsmethoden implementieren, die unter anderem für die Inhaltskontrolle (z.B. Filterung/Eingabenüberprüfung) genutzt werden können
- Methoden als privat deklarieren. Protected und public nur verwenden, falls notwendig
 - Protected und public Methoden müssen geschützt sein (Überprüfung der Eingabeparameter)
- Statische Klassenmitglieder vermeiden, da Zugriff auf den Inhalt aus allen Instanzen möglich

Basis-Programmiertechniken (2)

- Rückgabe von änderbaren Objekten vermeiden
 - z.B. Referenzen auf interne Arrays
- Änderbare Objekte, die vom externen Code übermittelt werden, sollen nicht direkt gespeichert werden
 - Objekt-Cloning verwenden
- Klasse String soll nicht zum Speichern sensibler Informationen (z.B. Kennworte) benutzt werden
 - Wegen Garbage-Collection werden sie nicht sofort freigegeben und können wieder benutzt werden
 - Stattdessen char[] verwenden und nach Verwendung überschreiben

Basis-Programmieretechniken (3)

- Klassen und Methoden nach Möglichkeit als final deklarieren
 - Erweiterungen/Modifikationen durch Angreifer werden vermieden
- Nach Möglichkeit nur final Methoden aus Konstruktoren heraus aufrufen
- Innere Klassen nicht benutzen
 - Innere Klassen werden beim Kompilieren als zugreifbar für alle Klassen im gleichen Package definiert
 - Die Klassenvariablen der umgebenden Klasse sind plötzlich nicht mehr privat

Basis-Programmiertechniken (4)

- Keine leeren catch-Blöcke verwenden
 - Auf Fehler wird nicht reagiert
 - Späteres „Füllen“ wird vergessen
- Standard-Bibliotheken für sicherheitsrelevante Funktionen verwenden
 - Keine neuen kryptografischen Verfahren ausdenken
- Erzeugung sicherer Zufallszahlen gewährleisten
 - `java.util.Random` nicht verwenden
 - stattdessen `java.security.SecureRandom`

Weitere Programmieretechniken (1)

- Nicht davon ausgehen, dass die Initialisierung immer stattfindet
 - Es existieren unterschiedliche Methoden, die Initialisierung zu umgehen
- Keine Klassenvergleiche auf Basis der Klassennamen verwenden (z.B. `if (obj.getClass().getName().equals("Foo"))`)
 - Stattdessen Vergleiche der Klassen mittels des `==` Operators verwenden
 - `if (a.getClass() == b.getClass())`

Weitere Programmieretechniken (2)

- Klassen als uncloneable realisieren, da durch Cloning-Mechanismus Objekte instanziiert werden können, ohne Ausführung eines Konstruktors
 - `public final void clone() throws java.lang.CloneNotSupportedException { throw new java.lang.CloneNotSupportedException(); }`
- Ist Cloning erforderlich...
 - Bei eigenen Methoden als final deklarieren
 - Bei fremden (Elternklasse):
 - `public final void clone() throws java.lang.CloneNotSupportedException { super.clone(); }`

Weitere Programmieretechniken (3)

- Klassen-Serialisierung vermeiden, da bei Serialisierung auch private Inhalte einer Klasse eingesehen werden können. Serialisierung durch folgende Methode nicht zulassen:
 - `private final void writeObject(ObjectOutputStream out) throws java.io.IOException { throw new java.io.IOException("Object cannot be serialized"); }`
- Ist Serialisierung erforderlich, alle Felder mit sensitiven Informationen (Kennwörter) bzw. lokalen Referenzen (z.B. Handles zu lokalen Ressourcen) als transient markieren
- Klassen als nicht deserializable implementieren, da die Deserialisierung auch für nicht serialisierbare Klassen möglich ist
 - `private final void readObject(ObjectInputStream in) throws java.io.IOException { throw new java.io.IOException("Class cannot be deserialized"); }`



Reverse Engineering und Code Obfuscation (Java)

Problematik

- Java Byte Code ist anfällig gegen Reverse Engineering
 - Dank vieler Informationen im Byte Code
- Code Obfuscation: Technologie, um das Reverse Engineering zu erschweren
 - Byte Code wird modifiziert. Als Ergebnis entsteht kaum lesbarer Spaghetti-Code
 - Ist jedoch kein Allheilmittel, nur der Aufwand für das Re-Engineering wird erheblich größer
 - Viele Code Obfuscators verfügbar

Reverse Engineering - Tools

- Verschiedene Decompiler verfügbar
 - Unterschiede in Funktionsweise, Lizenzierungsmodellen und Preisen
- Auswahl an bekannten Decompilern (keine komplette Liste):
 - Mocha, Jasmine, WingDis, DeJaVu, SourceAgain, Jad

Reverse-Engineering: Angriffsziele

- Problem: Java-Byte-Code muss ausgeliefert werden
- Dekompilieren liefert Informationen über
 - Kontrollfluss
 - Einprogrammierte Werte (Schlüssel, Initial-Werte,...)
 - Verarbeitete Eingaben (Versteckte Funktionen, Optionen)
 - Vorhandener Testcode (z.B. ohne Sicherheitsprüfungen)
 - Verwendete Mechanismen
 - zum Verschleiern von Daten
 - Prüfen von Lizenzen
- Vorbereitung für weitere Angriffe
- Wie kann Code Obfuscation helfen?

Layout Obfuscation

- Basiert auf ...
 - Umbenennung von Bezeichnern (vor allem Variablen- und Methodennamen)
 - Löschen von Debug-Informationen (Zeilen-Nummern, die vom Compiler eingefügt werden)
- Am weitesten verbreitete Technik
- Erschwert das Nachvollziehen, versteckt aber nicht die Programmlogik
 - Sicherheitsgewinn nicht erheblich

Layout Obfuscation - Beispiel

```

package test;

import java.util.*;

class Demo {

    private Vector buffer = new Vector();

    /**
     * Return the position of the specified String in the
     * buffer. Remove the String once if it has been
     * found.
     * Return -1 if the String isn't found. */
    int getStringPos(String string) {
        for(int counter=0; counter < buffer.size();
counter++) {
            String curString =
(String)buffer.elementAt(counter);
            if (curString.equals(string)) {
                buffer.remove(counter);
                return counter;
            }
        }
        return -1;
    }
}

```

```

package a;

import java.util.Vector;

class a {

    private Vector a new Vector();

    int a(String s) {
        for(int i = 0; i < a.size(); i++) {
            String s1 = (String)a.elementAt(i);
            if(s1.equals(s)) {
                a.remove(i);
                return i;
            }
        }
        return -1;
    }
}

```

Control Obfuscation

- Basiert auf Änderungen des Programm-Control-Flusses
 - Einführung zusätzlicher (auch ungültiger) Code-Abschnitte
- Ändert Programmablauf so, dass Nachvollziehen der Programmlogik kaum möglich ist
 - Bietet gute Sicherheit
 - Programme werden langsamer (etwa 2%, also nicht erheblich)
 - Programme werden größer durch Zusatzcode bzw. Code-Änderungen
- Durch Fehler im Code Obfuscator kann die fehlerfreie Funktionalität des Programms beeinflusst werden
 - Flow Obfuscation kann auch VM-/JIT-Fehler erzeugen/aufdecken

Control Obfuscation - Beispiel

```
c = a + b;
doSomething(c);
doOtherThings();
```

```
c = a + b;
if (methodWhichReturnsFalse()) {
doTotallyDifferentUselessThings();
} else {
doSomething(c);
doOtherThings();
}
```

```
if(i != 0) goto _L2; else goto _L1
_L1:
e != null ? e : (e =
a(b(">8\0171]2;\035!Z4)\035jO"));
_L2:
doSomething();
```

Data Obfuscation

- Basiert auf Änderungen des Datenmodells
 - Modifikationen der Vererbungsbeziehungen
 - Array-Restrukturierungen (Dimensionen)
 - Clone-Methoden (verschiedene Versionen von Methoden)
 - Aufspaltung der Variablen (eine boolesche Variable in zwei unterschiedlichen Variablen unterbringen)
 - String-Manipulationen
- Durch diese Änderungen ist Nachvollziehen des dekompilierten Codes kaum möglich; sehr mächtig
 - Bietet gute Sicherheit
 - Programme werden langsamer
 - Programme werden größer durch Zusatzcode bzw. Code-Änderungen

Code Obfuscator

- Code Obfuscator verschiedenster Typen verfügbar
 - Auch mit kombinierten Techniken
- Manche sind integrierbar in die Entwicklungsumgebungen
- Unterschiedliche Lizenzierungsmodelle und Preise
- Auswahl (keine komplette Liste):
 - 2LKit Obfuscator, Cloakware, CodeShield, Condensity, DashO Pro, Hashjava, Helseth Jobfuscator, HoseMocha, JAX, JCloak, Jobfuscate, JODE, JProof 1st Barrier , JShrink, JZipper, Obfuscate, Neil Aggarwal's Obfuscate, RetroGuard, Smokescreen, SourceGuard, The Marvin Obfuscator, Zelix KalssMaster

Fazit bzgl. Reverse Engineering

- Java Decompiler sind vorhanden
 - Einfach zu nutzen
- Code-Obfuscation kann helfen
- Relevant
 - wenn Java Programme an Dritte ausgeliefert werden
 - Wenn Angreifer Zugriff auf Java Programmcode haben
- Einzelfall-Entscheidung für/gegen Code-Obfuscation stets erforderlich



Anhang

Warum dieser Vortrag von uns?

- Unsere Erfahrung:

- mehrere Personenjahre in Forschungsprojekten zur sicheren Softwareentwicklung; derzeit gemeinsam mit Partnern wie SAP, Commerzbank, Universitäten, ...
- zahlreiche Schwachstellenanalysen für Softwarehersteller, nebst intensiver Feedbackzyklen mit den Entwicklern
- Erstellung von Anforderungs- und Designspezifikationen in mehreren großen Entwicklungsprojekten
- Erstellung von Guidelines zur sicheren Softwareentwicklung, mit Schwerpunkten Banking & Finance, sowie Webapplikationen
- Reverse Engineering und Gutachten von Sicherheitsfunktionen und Kryptomechanismen
- Implementierung von Sicherheitsfunktionen im Auftrag

Abschlussbemerkung

- die vorliegende Dokumentation wurde von EUROSEC erstellt im Rahmen des secologic Forschungsprojekts, Laufzeit 2005 und 2006, nähere Informationen unter www.secologic.org
- wir bedanken uns beim Bundesministerium für Wirtschaft für die Förderung dieses Projektes
- Anregungen und Feedback sind jederzeit willkommen, ebenso Anfragen zu Sicherheitsaspekten, die hier nicht behandelt werden konnten.

Copyright Hinweis

- Diese Folien wurden von EUROSEC erstellt und dienen der Durchführung von Schulungen oder Seminaren zum Thema Sichere Anwendungsentwicklung, mit Fokus Webapplikationen.
- Wir haben diese Folien veröffentlicht, um die Entwicklung besserer Softwareprodukte zu unterstützen.
- Die Folien dürfen gerne von Ihnen für eigene Zwecke im eigenen Unternehmen verwendet werden, unter Beibehaltung eines Herkunftshinweises auf EUROSEC.
- Eine kommerzielle Verwertung, insbesondere durch Schulungs- oder Beratungsunternehmen, wie beispielsweise Verkauf an Dritte oder ähnliches ist jedoch nicht gestattet.