Refactoring of Security Antipatterns in distributed Java components

Marc Schönefeld

Lehrstuhl für praktische Informatik
Otto-Friedrich-Universität Bamberg

12.7.2006

Agenda

1. Allgemeines
2. Kontext
3. Forschungsbeitrag
4. Anwendungsbeispiele
5. Schlussworte

Eckdaten der Dissertation

Status:

- Externe Dissertation, parallel zur Vollzeitbeschäftigung bei großem IT-Dienstleister

Betreuer:

- Prof. Dr. Guido Wirtz, Lehrstuhl für praktische Informatik, Otto-Friedrich-Universität Bamberg

Forschungszeitrahmen:

- Mitte 2002-laufend (erst WWU Münster, dann Otto-Friedrich Universität Bamberg)

Zwischenergebnisse:

- Präsentationen auf internationalen Konferenzen (u.a. RSA, Blackhat, DIMVA, AMCIS, D-A-CH)

Inhalt der Dissertation

Ziel:

- Methodik zur Verlagerung der Erkennung und Behebung von sicherheitsrelevanten Antipatterns in Java-Programmen in vorgelagerte Phasen des Software-Lebenszyklus

Umfang:

- Illustration und Früherkennung von antipattern-induzierten Sicherheitsproblemen
- Toolunterstützung für frühe Phasen der Software-Lebenszyklus,
  - Implementierung und
  - Komponenten-Integration (z.B. COTS)
- Anwendung auf javabasierte Verteilungsmiddleware, u.a. J2SE-Basis, J2EE-Server, JDBC-Provider, Java Media Framework

Motivation

- **Paradigmenwechsel** vom isolierten IT-System zum SOA-Provider
- Durch Internetdienste werden **verteilte Systeme allgegenwärtig**, von DCE über CORBA bis hin zu J2EE und .NET
- **Risiko-Lage ist erhöht**, denn IT-Systeme sind somit nicht ihrer angedachten Benutzerschaft ausgesetzt, sondern sind weltweit erreichbar (somit auch für Angreifer)
- Die Qualität der Erstellung und Konfiguration von IT-Systemen ist also nicht nur an der Erfüllung der funktionalen Aufgabe zu messen sondern auch bzgl. der Kriterien **“Überlebensfähigkeit, Sicherheit und Fehlertoleranz”** (Bishop 1997)

$\longrightarrow$ IT-Systeme muessen also **nicht nur korrekt und schnell**, sondern auch erreichbar, vertrauenswürdig und integer (kurz: sicher) sein

Verteilte Systeme und Sicherheit

The fallacies of distributed computing (Deutsch 1995)

- The network is reliable.
- Latency is zero.
- Bandwidth is infinite.
- **The network is secure**
- Topology doesn´t change.
- There is one administrator.
- Transport cost is zero.
- The network is homogeneous.

Umstieg auf verteile Systeme verursacht Probleme

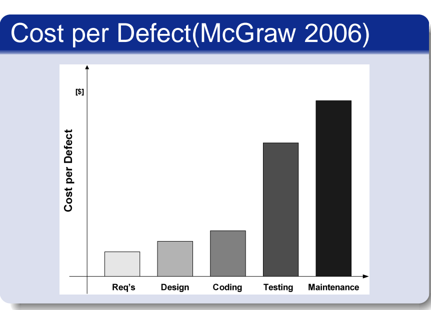
Nicht-funktionale Anforderungen (NFR) an Software wie u.a. Stabilität, Erweiterbarkeit, Interoperabilität und Sicherheit werden vernachlässigt.

Kosten nachgelagerter Securitymaßnahmen

- Sicherheitschecks oft nur notwendiges Übel kurz vor Shipping
- (externe) Penetrationstest und ”panic when attacked” **zu spät**
- **Kosten der Behebung von Schwachstellen steigen** mit jedem Phasenfortschritt im Entwicklungsprozess

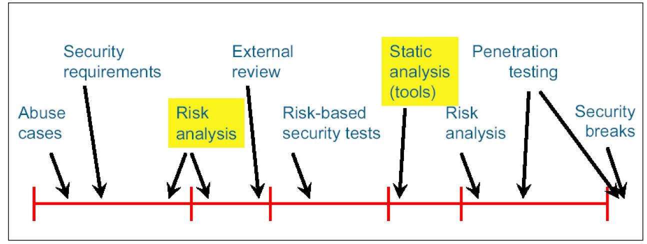
Ziel:

→ Security kein finaler Veredlungsschritt, sondern **ständiger Schritt** in iterativen Phasenmodellen (Lipner & Howard (2005), Lipson et al. (2002) und Beznosov & Kruchten (2004))



Sichere Produktentwicklung und Komponentenintegration

- McGraw (2004) fordert möglichst frühe Integration von Sicherheitsaspekten im gesamten SW-Entwicklungsprozeß , und
- **herstellerseitige** Beteiligung, die Hilfestellung sicherheitsspezifischen Integration von Komponenten, z.B. COTS (McGraw & Viega 1999) zur
- **Vermeidung** von “Fehlern in Konfiguration und Deployment” (OWASP Vulnerability A10) bietet



Funktional korrekte Programme und die Sicherheit

OWASP Top 10 (The Open Web Application Security Project 2004)

A1	Fehlerhafte Validierung der Eingabewerte
A2	Fehlerhafte Zugriffskontrolle
A3	Fehlerhafte Verwaltung von Benutzerkonten und Benutzersitzungen
A4	Kaperung von Sitzungen (aka XSS)
A5	Pufferüberläufe
A6	Kommando-Injektion
A7	Fehlerbehandlungsprobleme
A8	Unsichere Speicherung
A9	Verfügbarkeitsbedrohungen
A10	Fehler in Konfiguration und Deployment

Verwundbarkeiten

Nach Eckert (1998) teilen sich Verwundbarkeiten in drei Teilbereiche:

- **Konzeptionelle Fehler** entstehen in der Design-Phase, sind durch Modelleigenschaften vorbestimmt, Veränderung der Rahmenbedingungen, insb. der Bedrohungslage
- **Programmierfehler** in der Erstellungsphase, z.B. durch Skill- und Zeitmangel oder durch falsche Annahmen
  - **Sprachdesign:** z.B. fehlende Signalisierung Ganzzahl-Überlauf
  - **Implementierungsfehler:** z.B. in der Implementierung der Schutzfunktion der TCB (Aufruf privilegierter Funktionalität aus niedrigeren Schutzzonen)
  - **Algorithmen:** Unerwartete Auswirkungen von Komplexität in Basisklassen
- **Administrative Fehler** entstehen bei Einrichtung des Systems für produktiven Einsatz. Oft Kurzschaltung von Kontrollsystemen durch weitreichende Rechte (AllPermissions)

Antipatterns und Implementierungsfehler

Antipattern [Brown et al.,1998]

An **Antipattern** is a solution to common problems where the negative consequences of the solution exceed their benefits.

Brown et al. (1998):

- *AntiPatterns **clarify** problems for software developers, architects, and managers by identifying the smyptoms and consequences that lead to the dysfuncational software developement process*
- *AntiPatterns **convey** the motivation for change and the need for refactoring poor processes*
- *AntiPatterns are neceesary to gain an understanding of common problems faced by most software developers. Learning from other developer's successes and failures is valuable and neccessary. **Without this wisdom , AntiPatterns will continue to persist**"*

Struktur eines Antipattern

Strukturierungshilfe nach Brown et al. (1998)

- **Problem**
- **Hintergrund**
- **Kontext**
- **Einflussfaktoren**
- **Irreführende Annahmen**
- **Konsequenzen**
- **Symptome**
- **Korrigierte Lösung(Refactoring)**

Strukturverbesserungen durch Refactoring

Refactoring: Fowler (1999)

[. . .] is a set of techniques to identify and improving bad code, weeding out unnecessary code, and keeping the project as simple as possible.

- **Ziel:** Behebung von Antipattern zur Verbesserung des Verhältnisses zwischen den negativen Konsequenzen und den Vorteilen einer Problemlösung.
- **Ausmaß:**
  - **Kleine Refactorings** im Programmcode mit lokaler Auswirkung, z.B. Sicherheitspatch
  - **Große Refactorings** beeinflussen die Architektur (Beispiel: Java von 1.1 zu 1.2, Einführung von dynamischen Schutzdomänen).
- **Randbedingung funktionale Stabilität:** Refactorings betreffen nur die nicht-funktionale Anforderungsdimension von Programmen, d.h. keine Änderung des funktionalen Verhaltens.

Die Java2-Architektur

Java (Gosling et al. 2000)

Object-Oriented Programming Language aimed to provide network-centric applications

Kern-Sicherheitseigenschaften von Java(Gong 1999):

- Das **Laufzeitsystem** (JVM+Basisbibliotheken) bietet grundlegende TCB-Funktionen (z.B. Typsicherheit, Sichtbarkeit)
- **SecurityManager:** Least-Privilege-Rechtebündel für Applet-Sandbox oder Anwendungen ohne Vertrauensbeweis
- Monitoring basiert auf **Schutzzonen**-Prinzip (Stapelinspektion)
- **anpassbarer Rechte-Umfang** für Benutzercode (abh. von Vertrauensgrad und User-Identität)
- zusätzlich **Abschottungsmöglichkeiten** von Funktionalität und Daten durch JSSE, JAAS, GSSAPI, etc.

Java Sicherheitsmuster

Sun Security Code Guidelines (Sun Microsystems 2002)

- Careful usage of privileged code
- Careful handling of Static fields
- Reduced scope
- Careful selected public methods and fields
- Appropriate package protection
- If possible Use immutable objects
- Never return a reference to an internal array that contains sensitive data
- Never store user-supplied arrays directly
- Careful Serialization
- Careful use native methods
- Clear sensitive information

methodischer Forschungsbeitrag

- **Ansatz:** Der Entwicklungsprozess wird mit Methoden unterstützt , welche **vorausschauend** den Aspekt **Sicherheit** in Implementierung und COTS-Integration **fokussieren.**
- **Verlagerung** der Schwachstellenerkennung von Penetrationstests und Incidents in den Implementierungsprozess **zur frühzeitigen Vermeidung**
- zumindest bzw. bei COTS deren Schadenspotenzial zu erkennen und geeignet zu limitieren.

praktischer Forschungsbeitrag

Ergebnistypen in Form von praktischen Werkzeugen:

- Zur **Schadensvermeidung** setzt das *jDetect*-framework an: Mittels Binär-Analyse des ausgeführten Codes Problemmuster erkennen
- Zur **Schadensreduzierung** dient das *jChains*-framework: Auswirkungen von Problemmustern bei der Integration von Komponenten (z.B. COTS) durch Laufzeitanalyse erkennen und verringern

jDetect erkennt Antipattern während der Entwicklung

- Bytecode-Detektoren zur Kandidatensuche sicherheitsrelevater Problemmustern, u.a.
  - Verwundbare ReadObject-Methoden (OWASP A1,A9)
  - Propagation von Integer-Overflows (OWASP A1)
  - Input-propagation in Privilegierte Code-Blöcke (OWASP A1,A2)
  - Input-propagation (z.B. Randwerte) in Native Methoden (OWASP A1,A5,A6)

Beispiel

Das Sun JDK 1.4.2 hat über 8000 Klassen , jDetect reduziert für das Problem des Integer-Overflows die Kandidatenanzahl auf 15, diese können manuell durch Code-Inspection (Decompiler, Sourcecode) verifiziert werden (wg. falscher Positiver)

jChains analysiert Komponenten während des Deployments

Ziel:

- Verbesserung der Sicherheit im Deployment (OWASP A10) durch Ermittlung eines least-privilege Rechtesets

Arbeitsweise:

- Securitymanager-Interceptor wird in JVM registriert
- Rechteanfragen werden geloggt und Codesources zugeordnet,
- Zu oft wird beim Deployment von Java-Komponenten “AllPermissions”vergeben, stattdessen Ermittlung feingranularer Rechtezuordnung
- Erfolg abhängig von Coverage-Test-Strategie

Design:(vertikale) Verteilungsmiddleware:

- **Sender:** Ausführungs-JVM mit Interceptor-Plugin (”Abfangjäger”)
- **Empfänger:** Analyse-JVM für GUI
- **Kontrakt:** über OMG-IDL, somit auch jeder CORBA-kompatibler Empfänger anbindbar

Verwandte Forschungsbereiche

Die praktischen Teile der Arbeit nutzen:

- BCEL-Bibliothek (Dahm 2001) zur Implementierung der jDetect-Detektoren und das Findbugs-Framework (Hovemeyer & Pugh 2004) als Rahmenwerk zu deren Ausführung
- CORBA zur Entkopplung der Ausführungs- und Analysekomponente innerhalb von jChains

Verhaltensähnlichkeiten:

- Sandmark-Framework (Collberg & Townsend 2001) nutzt Bytecode-Detektoren für Software-Watermarking

Gemeinsamkeiten in der Zielsetzung:

- Howard & Leblanc (2002) stellen Methoden zur sicheren Software-Entwicklung für Microsoft-Produkte vor
- LSD, The last Stage of Delirium (2002) illustrieren Security Antipattern auf der JVM-Ebene
- Howard et al. (2005) zeigen generelle Kategorisierung für Verhaltensmuster in der Programmierung

Zusammenfassung

- **Kernpunkte:**
  - Sicherheit von Anwendungen und Middleware wird durch Antipattern in der Implementierung der TCB bedroht, daher steht der ausgelieferte Programmcode im Mittelpunkt denn:
  - OOA und OOD (UML, UMLSec) greift zu früh
  - Penetrationsansatz greift zu spät

  → Antipattern frühzeitig in den End-Ergebnissen des SW-Lebenszyklus erkennen und Refactorings durchführen
- **Methode:**
  - Bei Eigenerstellung **frühzeitige Erkennung** von Antipattern im ausgelieferten Code (mittels Bytecode-Detektoren)
  - Bei Drittkomponenten zusätzliches **Reverse Engineering** der Sicherheits-Bedürfnisse durch dynamische Laufzeitanalyse
- **Zielerreichung:** Verbesserung der Sicherheitseigenschaften von Middleware-Produkte (J2SE,J2EE,JDBC,. . .) durch Anwendung von *jchains* und *jdetect* (siehe Advisories)
- **weiteres Vorgehen:** Generalisierung auf .NET und Python

Appel & Wang (2002), JVM TCB: Measurements of the Trusted Computing Base of the Java Virtual Machine, Technical report, Princeton University.

Beznosov, K. & Kruchten, P. (2004), Towards agile security assurance.

Bishop, M. (1997), 'Information survivability, security and fault tolerance'.

blexim (2002), 'Basic integer overflows', *Phrack Magazine* **0x3c**.
URL: *http://www.phrack.org/phrack/60/p60-0x0a.txt*

Bosch, J. (2000), *Design and Use of Software Architectures : Adopting and Evolving a Product-Line Approach*, Addison-Wesley.

Brown, W. J., Malveau, R. C., III, H. W. S. M. & Mowbray, T. J. (1998), *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, John Wiley & Sons.

Collberg, C. & Townsend, G. (2001), 'Sandmark: Software watermarking for java'.
URL: *http://www.cs.arizona.edu/sandmark/*

Dahm, M. (2001), 'Byte Code Engineering with the BCEL API'.
URL: *http://citeseer.ist.psu.edu/dahm01byte.html*

Department of Defense (1985), 'Trusted Computer Systems Evaluation Criteria. DoD 5200.28STD'.
URL: *http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html*

Deutsch, L. P. (1995), 'Fallacies of distributed computing'.
URL: *http://java.sun.com/people/jag/Fallacies.html*

Eckert, C. (1998), Sichere, verteilte Systeme – Konzepte, Modelle und Systemarchitekturen, Habilitationsschrift, TU München.

Fowler, M. (1999), *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Object Technology Series, Addison-Wesley.

Gollmann, D. (1999), *Computer Security*, Wiley & Sons.

Gong, L. (1999), *Inside Java 2 Platform Security*, Addison-Wesley.

Gosling, J., Joy, B., Steele, G. & Bracha, G. (2000), *The Java Language Specification Second Edition*, Addison-Wesley, Boston, Mass.
URL: *http://citeseer.ist.psu.edu/gosling00java.html*

Hovemeyer, D. & Pugh, W. (2004), Finding Bugs is Easy, *in* 'OOPSLA'.
URL: *http://findbugs.sourceforge.net/docs/oopsla2004.pdf*

Howard, M. & Leblanc, D. E. (2002), *Writing Secure Code*, Microsoft Press, Redmond, WA, USA.

Howard, M., LeBlanc, D. & Viega, J. (2005), 19 deadly sins of software security.

Lipner, S. & Howard, M. (2005), 'The trustworthy computing security development lifecycle', *MSDN* .
URL: *http://msdn.microsoft.com/library/en-us/dnsecure/html/sdl.asp*

Lipson, H. F., McHugh, J., Mead, N. R. & Sledge, C. A. (2002), Life-cycle models for survivable systems, Technical report.
URL: *citeseer.ist.psu.edu/lipson02lifecycle.html*

LSD, The last Stage of Delirium (2002), 'Java and java virtual machine vulnerabilities and their exploitation techniques', Asia Black Hat Briefings 2002, Singapore.
Proof of concept codes available at
http://lsd-pl.net/projects/jvmvuln-1.0.2.tar.gz.

McGraw, G. (2004), 'Misuse and abuse cases: Getting past the positive', *IEEE Security & Privacy* pp. 90–92.

McGraw, G. (2006), *Software Security- Building Security In*, Addison-Wesley.

McGraw, G. & Viega, J. (1999), 'Why COTS software increases security risks'.
URL: *citeseer.ist.psu.edu/mcgraw99why.html*

Object Management Group (2001), *The Common Object Request Broker: Architecture and Specification*, 2.5 edn, Object Management Group.

Puder, A. & Römer, K. (2001), *Middleware für verteilte Systeme*, dpunkt-Verlag.

Schmidt, D., Stal, M., Rohnert, H. & Buschmann, F. (2000), *Pattern-Oriented Software Architecture*, Vol. Vol. 2: Patterns for Concurrent and Networked Objects, John-Wiley & Sons.

Schoenefeld, M. (2003*a*), 'J2ee 1.4 reference implementation: database component allows remote code execution'.
URL: *http://lists.seifried.org/pipermail/security/2003-December/000386.html*

Schoenefeld, M. (2003*b*), ‘Jboss 3.2.1: Remote command injection’.
URL: *http://cert.uni-stuttgart.de/archive/bugtraq/2003/10/msg00086.html*

Schoenefeld, M. (2003*c*), ‘[vulnwatch] java-applet crashes opera 6.05 and 7.01’.
URL: *http://archives.neohapsis.com/archives/vulnwatch/2003-q1/0063.html*

Schoenefeld, M. (2004), ‘Ibm cloudscape sql database (db2j) vulnerable to remote command injection’.
URL: *http://cert.uni-stuttgart.de/archive/bugtraq/2004/02/msg00142.html*

Schönefeld, M. (2003), Java Distributions and Denial-of-Service Flaws, Technical report, iDEFENSE Research Labs.
URL: *http://idefense.com/papers.html*

Sun Microsystems (2002), *Security Code Guidelines*.
URL: *http://java.sun.com/security/seccodeguide.html*

The Open Web Application Security Project (2004), ‘OWASP Top Ten

Most Critical Web Application Security Vulnerabilities'.

URL: *http://www.owasp.org/documentation/topten.html*