

Sichere Ausführung von „Untrusted Code“

Johannes Nicolai

Evaluation verschiedener Ansätze
und Einsatz an vier Fallbeispielen



Vorbemerkung

- Kurzzusammenfassung der Arbeit:
 - momentan 70 Seiten
 - nach Abschluss der Fallstudien ca. 100 - 120 Seiten
 - über 2000 Seiten Literaturrecherche
 - hier keine Präsentation konkreter Lösungen
 - Hinweise an geeigneter Stelle
 - Material verfügbar: <http://myhpi.de/~nicolai/>
-
-

Einleitung

- untrusted: nicht vollkommen vertrauenswürdig
 - Auftreten von Untrusted Code:
 - Plugins für Browser und Email Programme
 - verteiltes Rechnen (SETI), Grid
 - Benchmark- Systeme
 - Programmierwettbewerbe
 - Server Side Scripting bei Web- Hostern
 - ...
-
-

Einleitung

- Code läuft unter Benutzerrechten
 - Folgen / Risiken:
 - Daten einsehen, löschen, manipulieren
 - Ressourcenverbrauch
 - Denial of Service
 - Spam- Versand
 - ...
-
-

Einleitung

- Sicherheit:
 - Zugriff nur auf explizit genehmigte Ressourcen
 - Tatigung nur explizit genehmigter Aktionen
 - Schutz vor Denial of Service
- Sanktionen:
 - Verbot der angeforderten Aktion
 - Programmterminierung

Einleitung

- Prämissen:
 - kein domänenspezifisches Wissen
 - keine Kenntnis über korrekten Programmablauf
 - bereits existierende COTS Software
 - „Programm entsteht nicht auf der grünen Wiese“
 - keine Anpassung des Problems an die Lösung
-
-

Gliederung

1. Statische vs. dynamische Analyse
 2. Erfolgskriterien für statische Quellcodeanalyse
 3. Übertragung auf C / C++
 4. Erfolgskriterien für dynamische Analyse
 5. Klassifikation/ Evaluation dynamischer Ansätze
 6. Sichere faire Ausführung für RealTimeBattle
 7. Sicheres und faires Benchmark in Asparagus
 8. DCL
 9. Übungsbetrieb BSA
-
-

Statische vs. dynamische Analyse

- statische Analyse:
 - Überprüfung vor der Ausführung
 - einmalig
 - kein Overhead zur Laufzeit
 - evtl. schwierig zu bewerkstelligen
 - Ressourcenbelegung kaum kontrollierbar
 - „Blankoscheck“, der gerechtfertigt werden muss
 - häufig hybrider Ansatz
-
-

Statische vs. dynamische Analyse

- dynamische Analyse:
 - Überprüfung während der Ausführung
 - Overhead zur Laufzeit
 - flexibel und relativ einfach zu bewerkstelligen
 - Ressourcenbelegung sehr gut kontrollierbar
 - kein „Blankoscheck“, der gerechtfertigt werden muss
 - kein falscher Alarm
-
-

Erfolgskriterien für statische Quellcodeanalyse

- theoretische Existenz gesichert:
 - Transformation in einen endlichen Automaten (DFA)
 - Ressourcen sind Umgebung des DFA
 - Kategorisierung aller Zustände
 - Berechnung aller erreichbaren Zustände
 - Entscheidung über Sicherheit
 - Verfahren praktisch unmöglich
-
-

Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 1: Verfügbarkeit des Quellcodes
 - Geschäftsgeheimnis vieler Firmen
 - Lizenzen / NDAs
 - „security by obscurity“
 - Verbot von Änderungen am Quellcode / Garantie
 - Abhängigkeiten von weiterer Software
 - Quellen des Betriebssystems
-
-

Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 2: Unterstützung der Sprache
 - Dialekte
 - zum Teil sehr mächtige Konstrukte
 - Einschränkung von Konstrukten problematisch:
 - `cout << „Hello World“ << endl;` braucht:
 - Polymorphie
 - Zeigerarithmetik / Funktionszeiger
 - Templates
 - Überladung von benutzerdefinierten Operatoren
-
-

Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 3: Unterstützung von „echten“ Programmen (COTS Software)
 - häufig nicht nur Einsatz unbedenklicher Konstrukte
 - Nutzung von Standardbibliotheken
 - hohe Komplexität
 - dynamische Speicheranforderung / Nebenläufigkeit
 - nicht nur „pathologische“ Programme
 - meistens schon vor Methode etabliert
-
-

Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 4: Bewältigung der Komplexität
 - Kenntnis des Call Trees für Analyse notwendig
 - viele Variablen / komplizierte Berechnungen
 - evtl. Einsatz von Polymorphie / Funktionszeigern
 - potentielle Zustandsexplosion



Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 5: Schutz vor böswilliger Speicher manipulation
 - Angriff erfolgt nicht auf Programmebene
 - Ausführung von beliebigen Maschinencode durch
 - Buffer Overflows / Stack Smashing / Integer Overflows
 - Angriffe auf Heap- und Datensegment
 - Format String Angriffe
 - Notwendiger Perspektivwechsel:
 - Programm selbst ist potentieller Angreifer
-
-

Erfolgskriterien für statische Quellcodeanalyse

- Kriterium 6: Garantie für Prozessumgebung
 - Programm ist nicht autistisch
 - Angriff durch Delegation
 - Race Conditions
 - Betriebssystemschwächen

Erfolgskriterien für statische Quellcodeanalyse

- zusätzlich behandelt:
 - Proof Carrying Code:
 - sehr formale Herangehensweise
 - Prädikatenlogik (Vor- und Nachbedingungen)
 - lässt keine Interaktion mit Umgebung zu
 - sehr eingeschränkter Befehlssatz (8 Alpha Opcodes)
 - nur für kleine sicherheitskritische Algorithmen
 - weit fortgeschrittene Logikkenntnisse erforderlich
 - ... (siehe Kapitel 3)
-
-

Übertragung auf C / C++

- Erfüllbarkeit der 6 Kriterien für C / C++
 - Prämissen:
 - Quellcode der Applikation vorhanden
 - Quellen Bibliotheken und OS vorhanden
 - Nutzung der C / C++ Standardbibliothek
 - Polymorphie / Funktionszeiger
 - Templates
 - Exceptions
 - Zeigerarithmetik / Casting Operatoren
 - Präprozessordirektiven
 - Kriterien eins und drei erfüllt
-
-

Übertragung auf C / C++

- Problemfeld 1: Programmgröße und Interferenz

```
#include <stdio>  
using namespace std;
```

```
int main (int argc, char** argv) {  
    cout << "Hello world" << endl;  
}
```

Übertragung auf C / C++

- Problemfeld 1: Programmgröße und Interferenz
 - 29901 bzw. 17400 Zeilen nach Headerinklusion
 - teilweise inline Assembler
 - STL sehr mächtig, „komplizierter“ Code
 - Bibliotheken müssen mitanalysiert werden
 - Interferenz bei Namen / Makros / Spezialisierungen
 - Verbot von Systembibliotheken problematisch
 - Teilweise Erfüllung von Kriterium vier
-
-

Übertragung auf C / C++

- Problemfeld 2: Semantik
 - C++ sehr anspruchsvolle Semantik
 - nur wenige Compiler verstehen Semantik korrekt
 - Gründe:
 - eigene Operatoren
 - Typkonvertierung durch Operatoren und Konstruktoren
 - Templatespezialisierungen
 - Defaultargumente
 - Templatealgorithmen
 - eigene Allokatoren
 - Padding / Alignment
 - keine festgelegten Wertebereiche für Datentypen
 - Erfüllung von Kriterium zwei
-
-

Übertragung auf C / C++

- Problemfeld 3: Programmfluss
 - Programmpfade hängen von Variablenbelegung ab
 - Explosion des Zustandsraums
 - zusätzliche Probleme:
 - Polymorphie / Funktionszeiger
 - Signal- / Exithandler
 - Exceptions
 - Destruktoren
 - (Nebenläufigkeit / Interprozesskommunikation)
 - Kriterium vier nicht komplett erfüllbar
 - konservativer Ansatz: nicht erlaubte Funktionen dürfen nicht im Code auftauchen
-
-

Übertragung auf C / C++

- Problemfeld 4: Zeigerarithmetik
 - essentieller Bestandteil in C/ C++:
 - Parameterübergabe
 - Zeichenketten
 - Arrayzugriff
 - Polymorphie
 - effiziente Rohdatenmanipulation
 - falscher Speicherzugriff verletzt Kriterium fünf
 - Probleme bei Überprüfung:
 - absolute Gültigkeit bei variablen Einflussgrößen
 - zeitliche Gültigkeit
 - semantische Gültigkeit
-
-

```
union {
    struct {
        char x[8];
        char y;
    } z;
    int (*t) ();
} myObject;
char* pointer=myObject.z.x;
pointer[i]='A';
*(pointer+8)='B';
int hello () {
    printf("Hello world\n");
    return 0;
}
myObject.t=hello;
myObject.t();
pointer[1]='C';
myObject.t();
```

Übertragung auf C / C++

- Problemfeld 4: Zeigerarithmetik
 - zusätzliche Probleme:
 - Casting Operatoren
 - Placement New
 - schwer vorhersehbare Objektlebenszyklen
 - mehrfache Speicherfreigabe
 - Funktionen mit variabler Anzahl von Parametern:
 - `printf(„MeineAdresse%100d%7$n,25);`
 - bei `vsprintf` Missbrauch noch viel schwieriger zu entdecken
 - Erfüllung des sechsten Kriteriums:
 - statisch generell kaum machbar
 - alle Komponenten müssten berücksichtigt werden
-
-

Übertragung auf C / C++

- Schutz vor Speicher manipulation in C / C++
 - auf dynamischer / hybrider Ebene:
 - splint
 - Valgrind
 - CCMalloc
 - Electronic Fence
 - Purify Plus
 - MPatrol
 - CCured
 - Archerr Framework
 - ... (siehe Kapitel 4)

Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 1: Verfügbarkeit des Quellcodes
 - bei rein dynamischen Ansätzen nicht nötig
 - hybride Ansätze können davon profitieren
 - wichtiger sind Binärdateien
- Umformulierung des 1. Kriteriums:
 - „Verfügbarkeit der Binärdateien“
 - eventuell Modifikationserlaubnis

Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 2: Unterstützung der Sprache
 - bei rein dynamischen Ansätzen nicht nötig
 - hybride Ansätze können davon profitieren
 - wichtiger sind Binärdateien
- Umformulierung des 2. Kriteriums:
 - „Unterstützung des Betriebssystems“
 - „Unterstützung der Hardwarearchitektur“

Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 3: Zulassung von echten Programmen
 - Kriterium kann übernommen werden
 - industrielle Applikationen weiter lauffähig
 - wenig oder gar keine Fehlalarme

Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 4: Bewältigung der Komplexität
 - Betriebssystemzustand muss sehr gut bekannt sein:
 - aktuelles Verzeichnis
 - Flags von Deskriptoren
 - Wurzelverzeichnis
 - ...
 - Semantik der Systemrufe zum Teil diffizil:
 - Signale schicken mit `fcntl`
 - Dateideskriptoren mit `send` versenden



Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 5: Schutz vor böswilliger Speicher­manipulation
 - Prozesse haben virtuellen Adressraum
 - Speicherschutz der überwachenden Komponente
 - interner Ablauf des Programms kann sich ändern
 - nicht spezifizierte Zugriffe bleiben verwehrt

Erfolgskriterien für dynamische Quellcodeanalyse

- Kriterium 6: Garantie für Prozessumgebung
 - sämtliche Komponenten müssen kontrolliert werden
 - Race Conditions
 - „time of check, time of use“ (TOCTOU)
 - Systemruf: `open(„../test/mytest.dat“)`
 - Zugriff wird gestattet, echter Systemruf erfolgt
 - in der Zwischenzeit:
 - symbolischer Link
 - Verzeichniswechsel
 - Veränderung des Zeichenpuffers
 - Wechsel des Bezugssystems (`chroot`)

Klassifikation/ Evaluation dynamischer Ansätze

- Quellcodesubstitution
 - üblicherweise hybride Ebene
 - Abweichung vom normalen Pfad werden erkannt
 - fangen Speichermodifikationen ab
 - weitgehend plattformunabhängig
 - Einschränkungen:
 - Quellcode vorhanden, Programmiersprache unterstützt
 - viertes und sechstes Kriterium kaum erfüllt:
 - zu weit weg von Systemruffebene
 - keine Kontrolle über Umgebung
 - Zustand des Betriebssystems kaum feststellbar
 - untersuchte Lösungen: bereits genannt

Klassifikation/ Evaluation dynamischer Ansätze

- Binärcodemodifikation
 - Binärcode wird mit Prüfanweisungen instrumentiert
 - Systemrufe können zielgerichtet überprüft werden
 - Einschränkungen:
 - stark plattformabhängig
 - Änderung Binärcode erforderlich
 - Shell Skripte?
 - TOCTOU- Probleme (Kriterium 6)
 - Zustand des Betriebssystems schwer feststellbar
 - untersuchte Lösungen:
 - SASI
 - Software Based Fault Isolation
-
-

Klassifikation/ Evaluation dynamischer Ansätze

- Maschinencodeinterpreter
 - Binärcode wird interpretiert
 - Systemrufe können zielgerichtet überprüft werden
 - keine Änderungen von Komponenten erforderlich
 - Einschränkungen:
 - stark plattformabhängig
 - Shell Skripte?
 - TOCTOU- Probleme (Kriterium 6)
 - Zustand des Betriebssystems schwer feststellbar
 - untersuchte Lösungen:
 - Strata
 - DynamoRIO (hybrid)
 - Valgrind (hybrid)
-
-

Klassifikation/ Evaluation dynamischer Ansätze

- Intrusion Detection Systeme
 - siehe Kapitel 6
- Interpretersprachen / Safe Languages
 - Erfüllung der Kriterien eins bis fünf
 - Probleme mit Race Conditions
 - Sicherheitskonzepte in jeder Sprache verschieden
 - untersuchte Lösungen:
 - Java
 - .NET Code Access Security
 - Python / Perl / Ruby / PHP
 - Modula 3

Klassifikation/ Evaluation dynamischer Ansätze

- Härtung von bestehenden Code
 - schützen beteiligte Komponenten vor Angriffen
 - nur als Hilfsmechanismus zu sehen
 - schützen vor Speicher manipulationsangriffen
 - untersuchte Lösungen:
 - Securelib / SafeLib / libVerify
 - Format Guard
 - Stack Guard / StackShield / StackGhost
 - ExecShield
 - Pax / Adamantix
 - OpenWall / Owl

Klassifikation/ Evaluation dynamischer Ansätze

- System Call Interposition
 - jeder Systemruf muss genehmigt werden
 - least privilege access principle / whitelists
 - Aufgabenteilung:
 - system call interposition architecture (user / kernel mode)
 - policy engine (user / kernel mode)
 - können prinzipiell alle sechs Kriterien erfüllen
 - zusätzlicher Code im Kernel (Sicherheitsrisiko)
 - untersuchte Lösungen:
 - WHIPS, Remus, WindowBox
 - Janus, Consh, MapBox, Janus2, Ostia
 - Subterfuge, SLIC, Generic Software Wrappers
 - Systrace, Chakravyuha, Pea Pod, VServer
-
-

Klassifikation/ Evaluation dynamischer Ansätze

- Herkömmliche Betriebssystemmittel
 - sollten wo immer nur möglich vorgezogen werden
 - Einschränkungen:
 - die Rechtevergabe ist nicht granular genug,
 - Rechte nicht setzen nicht am Prozess an
 - Einführung von „Pseudonutzern“
 - Rechtebeschränkung von Prozessen freiwillig
 - viele Aufgaben erfordern Administratorprivilegien
 - Administrator selbst ist in keinster Weise eingeschränkt

Klassifikation/ Evaluation dynamischer Ansätze

- Domain Type Enforcement / ACLs
 - erweitern Betriebssystem um neues Rechtekonzept
 - Prozesse = Subjekte, Menge = Domain
 - Ressourcen = Objekte, Menge = Type
 - Matrix regelt Zugriff zwischen Domains und Types
 - Mandatory / Discretionary Access Control
 - können Kriterien eins bis sechs erfüllen
 - untersuchte Lösungen:
 - Security Enhanced Linux
 - RSBAC (castle, adamantix, kalladix)
 - LIDS (engardelinux)
 - Medusa, SubDomain
-
-

RealTimeBattle

- Robotersimulation als Programmierwettbewerb
- zu lösende Probleme:
 - Roboterprogramme sollen nur als Roboter agieren
 - faire Aufteilung von Ressourcen
- Lösung:
 - systrace
 - RSBAC



Asparagus

- Benchmark für Antwortmengenprogrammierung
 - zu lösende Probleme:
 - keine Manipulation von Systemdateien
 - „gleiche Bedingungen für alle“
 - Lösung:
 - Vserver
 - Betriebssystemmittel
 - Solaris Zones / Solaris Tasks
-
-

DCL

- verteilte Experimentierumgebung am HPI
 - zu lösende Probleme:
 - z. T. massive Echtzeitanforderungen
 - keine Beschädigung der Experimentieranordnung
 - Besonderheit: komplette Kontrolle über Problem
 - individuelle Lösung für jedes Experiment:
 - Code Access Security, Safe Languages, Safety Controller
 - Hardwarewatchdog, Codetransformation, große Puffer
-
-

BSA Übung

- Kontrolle der Übungsaufgaben
 - zu lösende Probleme:
 - keine Manipulation von Systemdateien
 - Nutzung erlaubter Programme einschränken
 - Lösung:
 - systrace- shell?
 - Betriebssystemmittel
 - RSBAC?
-
-

Noch Fragen?

