RE-TRUST

PoliTO Prototypes Overview

Paolo Falcarin, Stefano DiCarlo
Alessandro Cabutto, Nicola Garazzino, Davide Barberis

RE-TRUST meeting
Paris, Dec 2008
RE–TRUST Project

Many prototypes developed:

1. Mobile Code continuously replaced:
   a) Java Aspect on DynamicAOP – JVM
   b) Binary code on JVMTI interface – JVM5
   c) Binary code linked in EXE

2. Invariants checking

3. Control–Flow Checking
Prototype 1 a–b

Client (e.g. Chat Client) + Execution monitor

- Code Checker (agent)
- Tag Generator (agent)

CLIENT

Msg  Tag

SERVER

Agent manager

update

Server (e.g. Chat Server) + Tag checker

update
Integrity check

- Module contains
  - List of crypto hashes (each method)
  - Symmetric key
- Keyed hash recomputed each time a method is called
- Hash compared with “good” copy
- New Module checks previously deployed ones
Prototype 1a–b

- Execution interception
  - Module calculates proof
  - Seamless replacement at run–time
  - Slow start–up

- Transparent tag insertion
  - Call to socket write are intercepted and data buffer is tagged

- Client code:
  - Its image is checked in memory (JVMTI)
  - can be sandboxed (AOP)
 Prototype 1c

- The application is deployed incomplete
- Some binary code blocks are downloaded and linked in memory at start-up
- The memory layout is decided by the server, and it changes at every run
- Useful to defeat static analysis
Program Instrumentation step

- PE header patch
  - make code segment writable at runtime;
- Disassembling the executable collecting information about control-flow
  - Instructions calls, jump, ret;
- Functions “to be protected” are purged
  - They won’t be available in client application;
- Patching of calls/jumps referring to purged functions:
  - they will now point to scheduler()
Runtime

• When scheduler is invoked → it send caller address to trusted node
• The TN uses its lookup tables to find out if the caller needs some purged code to be sent to the UN
• The scheduler in UN
  • receives the necessary code
  • Allocate it in memory
  • then executes it
Remote Control Flow Checking

- Split the program integrity verification among the untrusted and the trusted node:
  - Program execution performed on the untrusted node
  - Control flow validation performed on the trusted node
Remote Control Flow Checking

- Basic flow:
  - The target application collects information (traces) about executed instructions
  - Traces are transmitted from the untrusted node to the trusted node
  - The trusted node validates the control flow of the application
  - Any violation is detected as an attack
The trusted node is in charge of:

- Monitoring the flow of instructions received from the untrusted node (correct sequence of basic blocks)
- Validating the checksum of each basic block (correct instructions opcode)
Invariants Prototype: Car Race

- Language: C++
- Graphic Library: Open GL
- IDE: Visual Studio 2003
- One Server, 2 players
Prototype: Car Race

- There are 10 check points.
- Every check points has a particular picture with DRM
Car Race: Security

- Security:
  - Secure Protocol
  - Invariants Checks
  - Mobile Code
  - Mutual Authentication

- Invariants Check
  - To protect algorithm
  - To protect DRM
Car Race: Security

- Mobile Code
- Mutual Authentication → A client needs the server authentication, in fact without this “ACK”, the trusted platform uses a wrong key to crypt information.
Initial Analysis
Open issues with mobility

- How to protect mobile modules?
  - Obfuscation

- How often a module is replaced?
  - It depends on time needed to understand it and implement an attack

- How to measure this Time–2–Break–It?
  - Metrics
  - Empirical Evaluation
Obfuscation Metrics

- Source Code complexity
  - Potency and other metrics (Collberg et al.)
  - Depth of Parse Tree (Goto et al.)
  - DeObfuscation Time (Udupa et al.)

- Binary Code complexity
  - Confusion Factor (Linn et al.): % of code that cannot be disassembled

- Compare Obfuscations (Anckaert et al.)
  - Code & control flow metrics
  - Data and data-flow metrics
Empirical Evaluation of Obfuscation

- **Complexity of reverse engineering binary code**
  - Asking a group of 10 students to perform static analysis, dynamic analysis and change tasks on several C (compiled) programs.
  - They found that the subjects’ ability was significantly correlated with the success of reverse engineering tasks they had to perform.

- **Complexity of Java id-renaming source code obfuscation:**
  - Controlled experiment of Master and Ph.D students to crack 2 apps (one clear, one obfuscated)
  - Calculate statistical effect size of the applied obfuscation
References

References