CSE331: Introduction to Networks and Security

Lecture 23
Fall 2006
Talk announcement

Kostas G. Anagnostakis
Systems & Security Department
I2R, Singapore
http://s3g.i2r.a-star.edu.sg/

Proximity Breeds Danger: Wildfire Worms, Wireless Phishing and Citizen Tracknets in Metro Wi-Fi Networks

Friday, November 3, 2006
Wu & Chen, 101 Levine Hall
3:00 pm - 4:30 pm
Recap

• Faulty Software
  – Buffer overflows

• Malicious Software
  – Viruses
  – Worms
  – Trojan Horses
  – Insider Attacks

• Today:
  – Current research in *language-based* security
Proof-Carrying Code (PCC)
Problem: faulty/malicious software

• What shall we trust?
  – Source code
  – Compiler
  – Virtual machine / run-time library?

• Trusted Computing Base
  – TCB is huge on real systems
  – Minimizing the TCB can improve software quality

• How to minimize TCB?
  – Must be easy to deploy
Existing Approach: Virus Scanners

- Virus Scanners?
  - e.g., McAfee, Norton, etc.
  - perhaps the most commercially effective tool.
  - TCB is small: only check the binary code

- Works for:
  - Previously seen malicious code.

- Can’t deal with:
  - Software bugs
  - Unknown viruses

- Other limitations:
  - virus kits make it easy to disguise a virus.
  - not clear that it scales over time.
Existing Approach: Code Signatures

- Digital Signatures of Code?
  - e.g., Verisign, Authenticode, MS device drivers
  - TCB is also small

- Works for:
  - Code that needs to be trusted

- Can’t deal with software bugs
  - Well-intentioned people make “bad” code

- Can we do better?
Language-based Security

• Use compiler & programming language technology to improve security.

• Static approaches:
  – Compiler performs type checking
  – Reject a bad program **before** it can run

• Dynamic approaches:
  – Code instrumentation: automatic insertion of safety checks in machine code
  – Software bug discovered at run time, but no damage is done
Type checking

• Strongly typed languages:
  – Java, Pascal, Ada, ML, etc.
  – Compiler does type checking and reject faulty programs

• Security guarantee:
  – Well-typed programs doesn’t go wrong (no buffer overflows)

• Problem: TCB is huge
  – Does the type system make sense?
  – Compiler has no bugs?
Example: Java Bytecode

- Verify the bytecode at the consumer
  - Essentially, type checking
- Pros: simple, cost effective
- Cons: Large TCB:
  - commercial, optimizing JIT: 200,000-500,000 LOC
  - when is the last time your favorite software company wrote a bug-free 200,000 line program?
- Limitation: Java specific
  - What about real machine code?
Proof Carrying Code

[Necula & Lee ‘97, Morrisett ‘98, Appel...]

• Verify a provided proof of program security
  – Meaning of the proof connected to meaning of program (unlike signatures)
  – Up to code producer to generate proof
  – Consumer only has to check the proof

• Verifier is small
  – 3000 LOC
PCC: An Analogy

Legend:  

- code
- proof
PCC Advantages

- Reduces the TCB
  - Verification is simpler/faster than proof generation.
  - Consumer is independent of how the proof is generated ⇒ compiler not trusted.
- Tamperproof
  - Changing the proof or program is either (1) detected or (2) proven to be OK.
- No cryptography, no trusted 3rd party
- No run-time overhead
  - Static checking
PCC Engineering Challenges

• Where do you get the proof?
  – Programmer & compiler
  – Automated techniques needed

• Dealing with formal proofs
  – Must be machine checkable
  – Naive encoding of proofs of program properties are very large.
    • Careful engineering reduces overhead

• Touchstone Compiler [Necula & Lee]
  – Java to Intel x86 assembly language
  – Enforces Java’s security policy without byte code interpreter or large trusted JIT
PCC and security policies

• PCC is not limited to safety policies
  – type safety \(\Rightarrow\) no crashes
  – in principle, PCC can enforce any policy
  – ... but how to describe other policies?

• Programming languages with facilities for implementing specific policies
  – Confidentiality
    • protect secrets
  – Integrity
    • prevent tampering
Language-based Information-flow Security
Information Flow Policies

[Myers, Zdancewic, Zheng, Chong, Nystrom]

• Lots of confidential info.
  – passwords, e-mail, financial data, medical data, business transactions, ...

• Example policy for a untrusted code:
  – Code can read local files
  – Code can send data to the network
  – However, no information read from the local files can be sent to the network

• Existing mechanisms are not enough
  – OS doesn’t provide fine grained control
  – Cryptography not always the solution
  – Not “end-to-end” solutions
Jif = Java + Information Flow

- Idea: write information-flow policies directly in the programs
  - Policies as data types
- Compiler checks the information flow throughout the program
Security Policies in Jif

- Confidentiality labels:
  \[ \text{int}\{\text{Alice}::\} \ x; \]  
  "Alice's private int"
  \[ \text{int}\{\text{Alice:Bob}\} \ y; \]  
  "Alice permits Bob as reader"

- Integrity labels:
  \[ \text{int}\{\text{*:Alice}\} \ z; \]  
  "Alice trusts z"

- Combined labels:
  \[ \text{int}\{\text{Alice: ; *:Alice}\} \ w; \]  
  (Both)

\[
\begin{align*}
\text{int}\{\text{Alice}:\} & \ a1, a2; \\
\text{int}\{\text{Bob}:\} & \ b; \\
\text{int}\{\text{*:Alice}\} & \ c;
\end{align*}
\]

Insecure
\[
\begin{align*}
a1 & = b; \\
b & = a1; \\
c & = a1;
\end{align*}
\]

Secure
\[
\begin{align*}
a1 & = a2; \\
a1 & = c;
\end{align*}
\]
Implicit Information Flow

int{Alice:} X;
int{Bob:} Y;
...
if (X > 0) then {
    Y = 1;
} else {
    Y = 0;
}
//… computation that uses Y…
Information Confidentiality
Advantages of Jif

• Explicit information-flow policies
  – compiler checks program for compliance
• Finer granularity than OS: enforces rich, programmable policies
  – e.g. “Medical data should not be sent to the public printer.”
  – e.g. “Financial data should be encrypted before being transmitted over the Internet.”
• End-to-end security:
  – Policy specified at “end points” (input & output interface)
  – Policy enforced throughout the entire system
• Similar technology already used:
  – Taint-checking mode in Perl
    • Prevents “bad” data from being used inappropriately
Summary

• Proof-Carrying Code (PCC):
  – Minimal trusted computing base
  – Can be used to enforce safety policies
  – Can also be used for other policies…

• Language-based Information-flow Security:
  – Fine-grained, end-to-end policies
  – Enforced in programming languages (Jif)