

CIS/TCOM 551

Computer and Network Security

Spring 2005

Lecture 9

Announcements

- Midterm 2 will be held Thursday: March 31st
 - In class
 - Format similar to first midterm: short answer & problem solving
 - Topics: Principles of secure design, software security, Java stack inspection, worms & viruses, epidemic model, malware detection & prevention

- Final project will be handed out soon.
 - Group based
 - Due last day of classes: Friday, April 22nd.

Today's Agenda

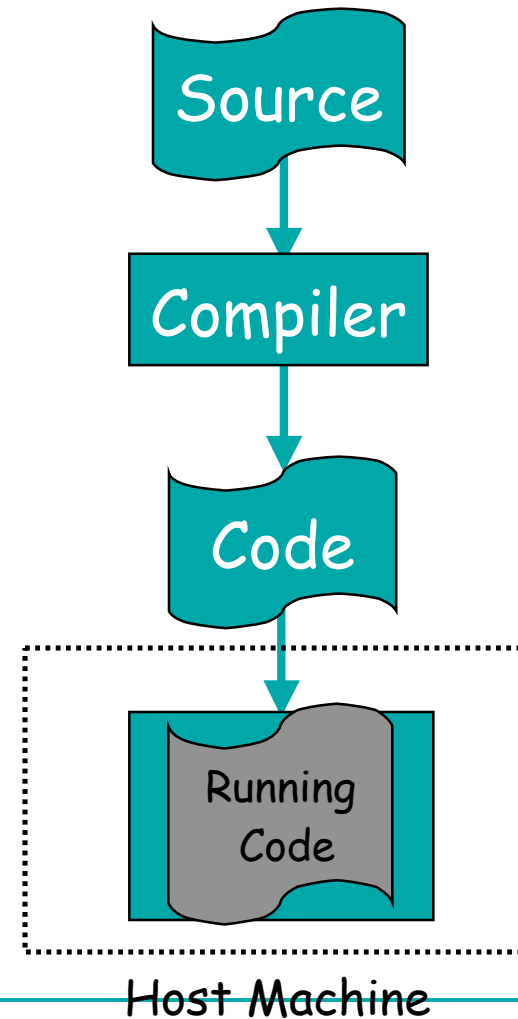
- Revisit code security
 - Compare & contrast existing mechanisms
 - Look at what's being developed in the research community
- Review

Malicious Code: Static Solutions

- Don't write code in C
 - Use a safe language instead (Java, C#, ...)
 - Not always possible (low level programming)
 - Doesn't solve legacy code problem
- Link C code against safe version of libc
 - May degrade performance unacceptably
- Software fault isolation / code instrumentation
 - Instrument executable code to insert checks
 - StackGuard / PointGuard
 - Pay a performance penalty
- Program analysis techniques
 - Examine program to see whether “tainted” data is used as argument to strcpy.
 - Possible to eliminate dynamic checks

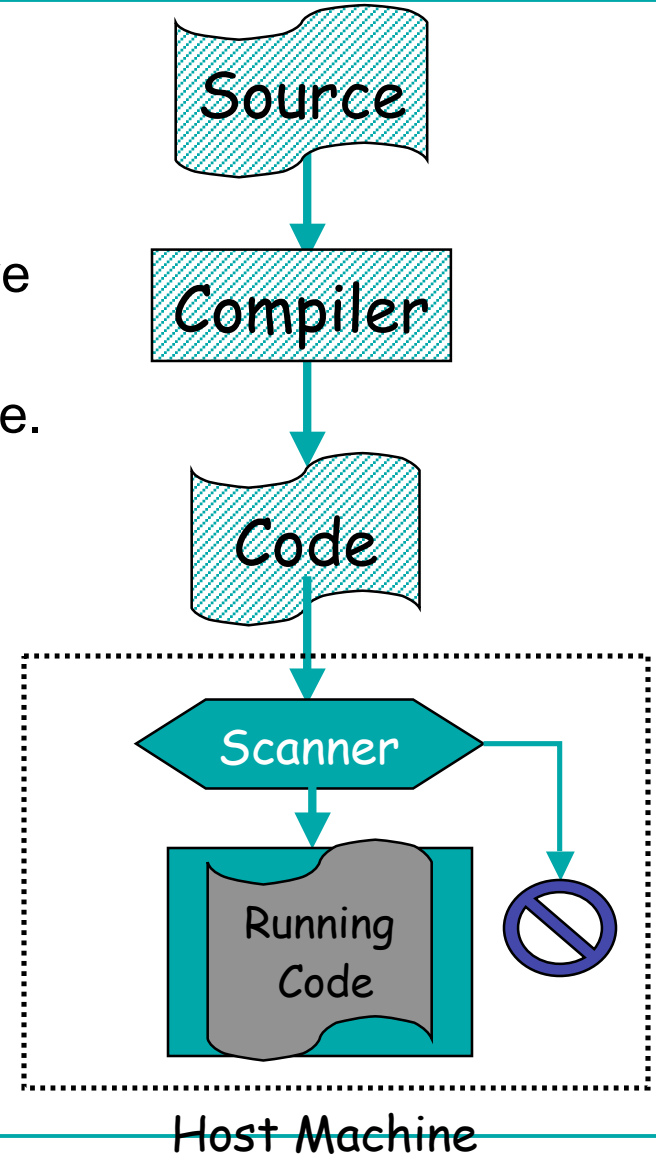
Software Deployment Architecture

- Trusted Computing Base
 - Becomes huge when software is run on many, many hosts
- Minimize TCB:
 - Ensure the quality of the software
- Must be cheap, easy to deploy
 - Otherwise won't be adopted



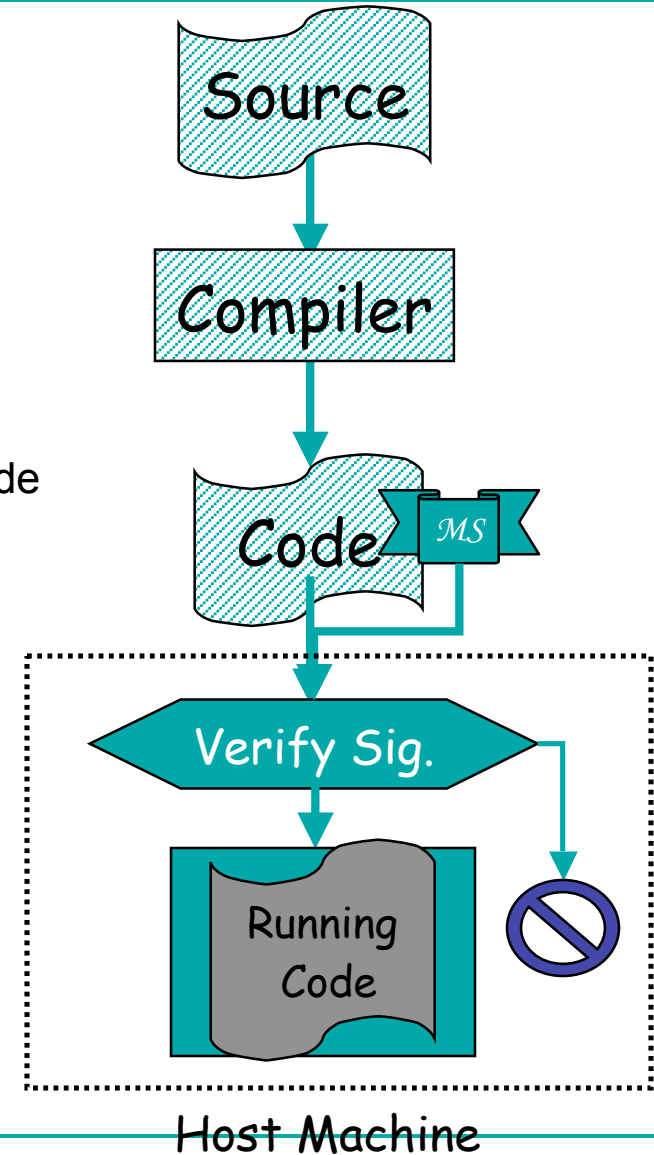
Existing Approach: Virus Scanners

- Virus Scanners?
 - e.g., McAfee, Norton, etc.
 - perhaps the most commercially effective tool.
 - only works for previously seen bad code.
 - virus kits make it easy to disguise a virus.
 - not clear that it scales over time.
- Not a complete solution



Existing Approach: Signatures

- Digital Signatures of Code?
 - e.g., Verisign, Authenticode, MS device drivers
 - bad assumption: signature implies “good”
 - keys may be stolen
 - “good” for what context?
 - even well-intentioned people make “bad” code
 - bad assumption: you can sue the signer
- Not a complete solution
- Can we do better?



Language-based Security

- Use compiler & programming language technology to improve security.
- Before the program runs
 - Static type systems (Java/C#)
- During the program execution
 - Array bounds checks
 - Stack inspection
- After the program runs
 - Bug report features
 - Auditing



Protects the code **consumer**.



Helps the code **producer**



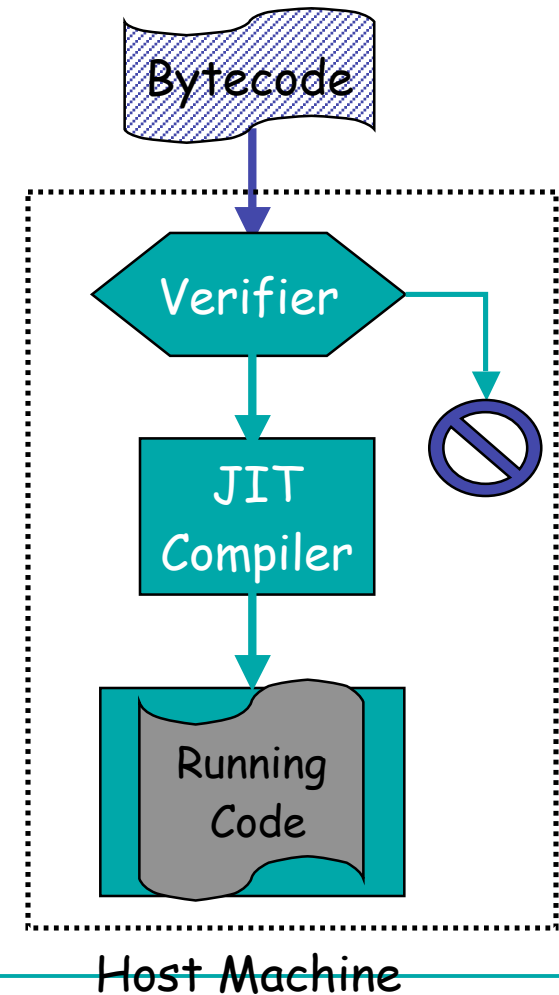
Protects the code **consumer**.



Helps the code **producer**

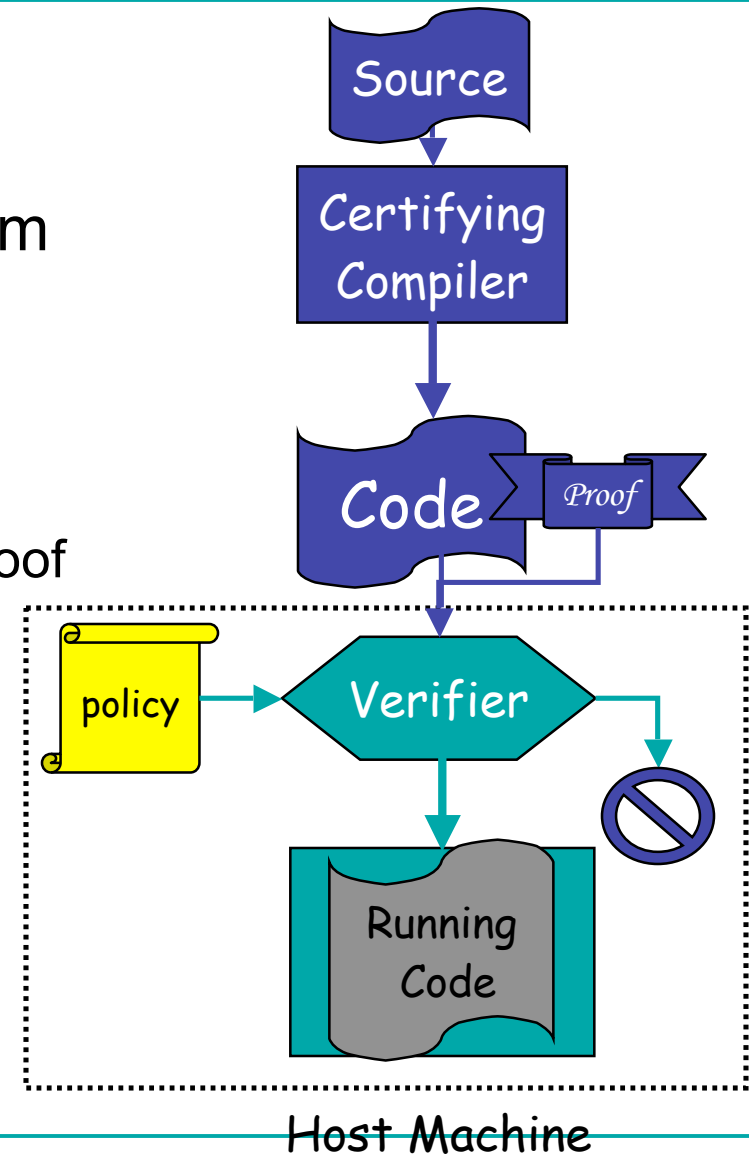
Java Bytecode

- Verify the bytecode at the consumer
- Pro: Simple, cost effective
- Con: 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?
- Con: Java specific policy?
- Pro: Programming model does support some access control features
 - Stack inspection



Proof Carrying Code

- Necula & Lee, Morrisett et al.
- 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*
 - Originally < 3000 LOC
 - Now < a few hundred LOC



PCC Advantages

- Reduces the TCB
 - Verification is simpler/faster than proof generation.
 - Consumer is independent of how the proof is generated \Rightarrow 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

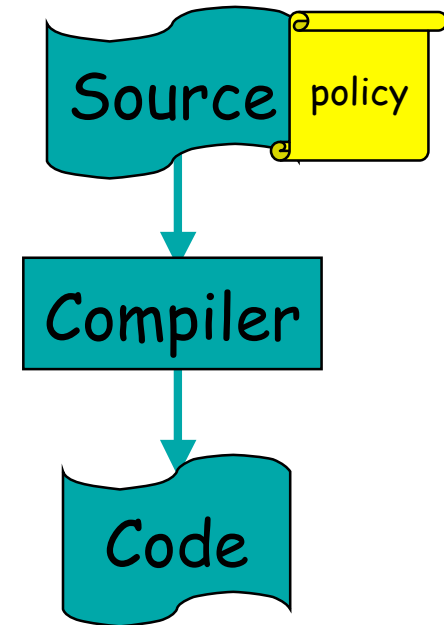
Security-oriented Languages

- PCC doesn't address policy
 - type safety \Rightarrow no crashes
 - in principle, can enforce any policy
 - ... but how to describe the policy?
- Programming languages with facilities for implementing specific policies
 - Confidentiality
 - protect secrets
 - Integrity
 - prevent tampering
 - Availability
 - ensure legitimate use succeeds

Jif = Java + Information Flow

[Myers, Zdancewic, Zheng, Chong, Nystrom]

- Problem: Lots of confidential info.
 - passwords, e-mail, financial data, medical data, business transactions, ...
- Existing technology essential, but...
 - OS doesn't provide fine grained control
 - Cryptography not the solution
 - Not “end-to-end” solutions
- Philosophy: improve security, do not try to eliminate covert channels
 - Modern take on MLS security



Security Policies in Jif

- Confidentiality labels:
`int{Alice:} x;` "Alice's private int"
`int{Alice:Bob}y;` "Alice permits Bob as reader"
- Integrity labels:
`int{*:Alice} z;` "Alice trusts z"
- Combined labels:
`int{Alice: ; *:Alice} w;` (Both)

```
int{Alice:} a1, a2;  
int{Bob:} b;  
int{*:Alice} c;
```

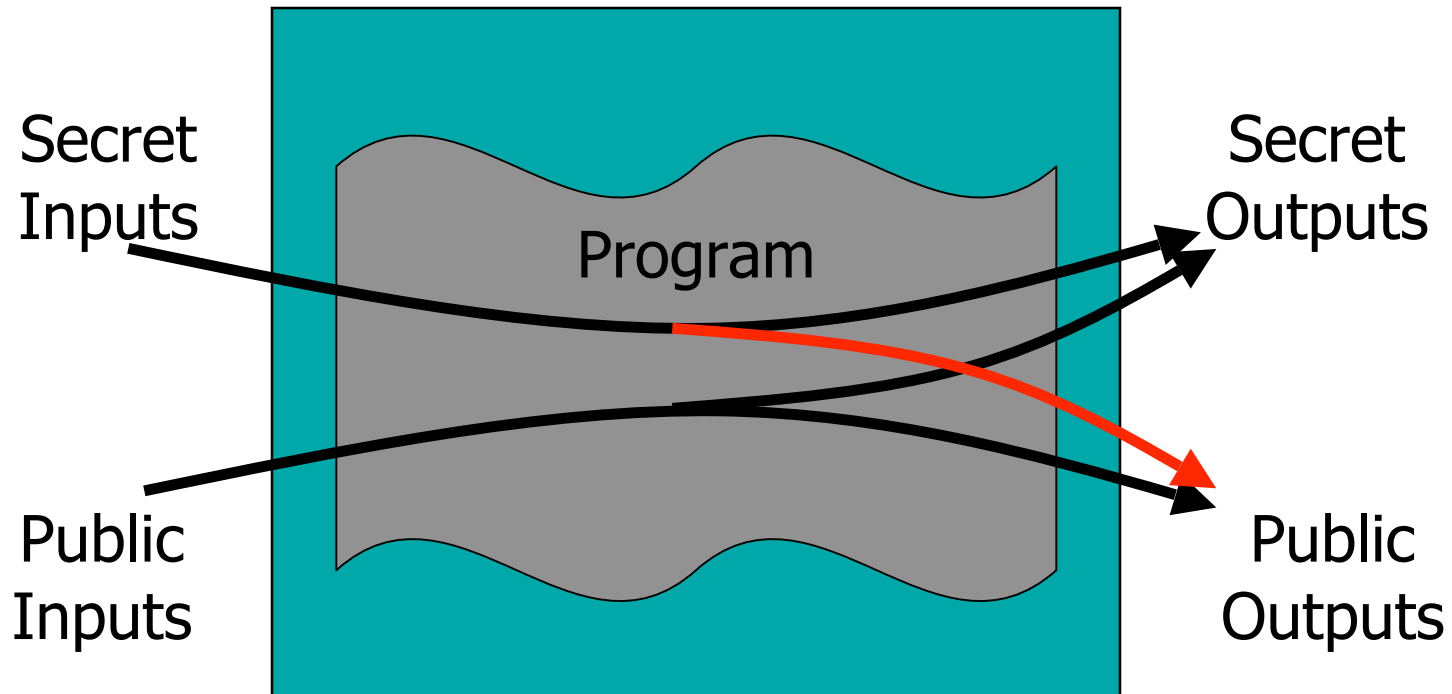
Insecure

```
a1 = b;  
b = a1;  
c = a1;
```

Secure

```
a1 =  
a2;  
a1 = c;
```

Information Confidentiality



Jif Advantages

- 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.”
- Permits end-to-end security
- Similar technology already or soon to be used:
 - Perl: Prevents “bad” data from being used inappropriately (lightweight MLS)
 - Microsoft e-mail will control dissemination

Jif Disadvantages

- Significantly more complex to develop software
 - Programmer has to be aware of security policies and develop the software accordingly.
- Policy is intermingled with the software
 - Usually a good idea to separate specification from implementation.
- Doesn't work well for malicious code setting
 - Mainly helps the code producer
 - Could imagine using PCC approach to help protect code consumer